Building Windows 8 Metro, Web, and Desktop Applications for the .NET 4.5 Framework

Programming C# 5.0

Early Release

O'REILLY®

Ian Griffiths

1

Introducing C#

The C# programming language (pronounced 'see sharp') can be used for many kinds of applications, including web sites, desktop applications, games, phone apps, and command line utilities. C# has been center stage for Windows developers for about a decade now, so when Microsoft announced that Windows 8 would introduce a new[1] style of application, optimized for touch-based interaction on tablets, it was no surprise that C# was one of the four languages to offer full support from the start for these *Metro style applications*, as they're called, (the others being C++, JavaScript, and Visual Basic).

Although Microsoft invented C#, the language and its runtime are documented by the standards body ECMA, enabling anyone to implement C#. This is not merely hypothetical. The open source Mono project at http://www.mono-project.com/ provides tools for building C# applications that run on Linux, iOS and Android.

Why C#?

Although there are many ways you can use C#, other languages are always an option. Why might you choose C# over these? It will depend on what you need to do, and what you like and dislike in a programming language. Speaking for myself, I find that C# provides considerable power and flexibility, and works at a high enough level of abstraction that I don't expend vast amounts of effort on little details not directly related to the problems my programs are trying to solve. (I'm looking at you, C++.)

Much of C#'s power comes from the range of programming techniques it supports. For example, it offers object-oriented features, generics, and functional programming. It supports both dynamic and static typing. It provides powerful list- and set-oriented features thanks to LINQ. The most recent version of the language adds intrinsic support for asynchronous programming.

[1] New to Windows, at any rate.

Some of the most important benefits of using C# come from its runtime, which provides services such as security sandboxing, runtime type checking, exception handling, thread management, and perhaps its most important feature, automated memory management. The runtime provides a garbage collector that frees developers from much of the work associated with recovering memory that the program is no longer using.

Of course, languages do not exist in a vacuum—high quality libraries with a broad range of features are essential. There are some elegant and academically beautiful languages that are glorious right up until you want to do something prosaic such as talking to a database, or determining where to store user settings. No matter how strong a set of programming idioms a language offers, it also needs to provide full and convenient access to the underlying platform's services. C# is on very strong ground here, thanks to the .NET Framework.

The .NET Framework encompasses both the runtime and the libraries that C# programs use on Windows. The runtime part is called the *Common Language Runtime* (usually abbreviated to CLR) because it supports not just C#, but any .NET language. Numerous languages can run in .NET. Microsoft's development environment, Visual Studio, provides Visual Basic and F#, for example, and there are open source .NET-based implementations of Python and Ruby (called IronPython and IronRuby). The CLR has a *Common Type System* (CTS) enabling code from multiple languages to interoperate freely, which means that .NET libraries can usually be used from any .NET language—F# can consume libraries written in .NET, C# can use Visual Basic libraries, and so on. The .NET Framework includes an extensive class library. This library provides wrappers for many features of the underlying operating system, but it also provides a considerable amount of functionality of its own. It contains over 10,000 classes, each with numerous members.

> Some parts of the .NET Framework class library are specific to Windows. There are library features dedicated to building Windows desktop applications, for example. However, other parts are more generic, such as the HTTP client classes, which would be relevant on any operating system. The ECMA specification for the runtime used by C# defines a set of library features which are not dependent on any particular operating system. The .NET Framework class library supports all these features of course, as well as offering Microsoft-specific ones.

The libraries built into the framework are not the whole story—many other frameworks provide their own .NET class libraries. SharePoint has an extensive .NET API, for example. And of course, libraries do not have to be associated with frameworks. There's a large ecosystem of .NET libraries, some commercial and some free and open source. There are mathematical utilities, parsing libraries, and user interface components to name just a few.

Even if you get unlucky and need to use an OS feature that doesn't have any .NET library wrappers, C# offers various mechanisms for working with older style APIs such as Win32 and COM. Some aspects of the interoperability mechanisms are a little clunky, and if you need to deal with an existing component, you might need to write a thin wrapper that presents a more .NET-friendly face. (You can still write the wrapper in C#. You'd just be putting the awkward interop details in one place, rather than letting them pollute your whole codebase.) However, if you design a new COM component carefully, you can make it straightforward to use directly from C#. Windows 8 introduces a new

style of API for writing Metro style tablet applications, an evolution of COM called *WinRT*, and unlike interop with older native Windows APIs, using WinRT from C# feels very natural.

In summary, with C# we get a strong set of abstractions built into the language, a powerful runtime, and easy access to an enormous amount of library and platform functionality.

Why Not C#?

To understand a language, it's useful to compare it with some alternatives, so it's worth looking at some of the reasons you might choose some other language. Its nearest competitor is arguably Visual Basic, another native .NET language which offers most of the same benefits as C#. The choice here is mostly a matter of syntax. C# is part of the C family of languages, and if you are familiar with at least one language from that group (which includes C, C++, Objective C, Java, and JavaScript) you will feel instantly at home with C#'s syntax. However, if you do not know any of those languages, but you are at home with pre-.NET versions of Visual Basic, or with the scripting variants such as Microsoft Office's Visual Basic for Applications (VBA), then the .NET version of Visual Basic would certainly be easier to learn.

Visual Studio offers another language designed specifically for the .NET Framework, called F#. This is a very different language from C# and Visual Basic, and it seems to be aimed mostly at calculation-intensive applications such as engineering, and the more technical areas of finance. It is primarily a functional programming language, with its roots firmly in academia. (Its closest non-.NET relative is a programming language called OCaml, which is popular in universities, but has never been a commercial hit.) It is good for expressing particularly complex computations, so if you're working on applications that spend much more of their time thinking than doing, F# may be for you.

Then there's C++, which has always been a mainstay of Windows development. The C++ language is always evolving, and in the recently published C++11 standard (ISO/IEC standard 14882:2011, to use its formal name), the language gained several features that make it significantly more expressive than earlier versions. It's now much easier to use functional programming idioms, for example. In many cases, C++ code can provide significantly better performance than .NET languages, partly because C++ lets you get closer to the underlying machinery of the computer, and partly because the CLR has much higher overheads than the rather frugal C++ runtime. Also, many Win32 APIs are less hassle to use in C++ than C#, and the same is true of some (although not all) COM-based APIs. For example, C++ is the language of choice for using the most recent versions of Microsoft's advanced graphics API, DirectX. Microsoft's C++ compiler even includes extensions that allow C++ code to integrate with the world of .NET, meaning that C++ can use the entire .NET Framework class library (and any other .NET libraries). So on paper, C++ is a very strong contender. But one of its greatest strengths is also a weakness: the level of abstraction in C++ is much closer to the underlying operation of the computer than in C#. This is part of why C++ can offer better performance, and is able to consume certain APIs more easily, but it also tends to mean that C++ requires considerably more work to get anything done. Even so, the tradeoff can leave C++ looking preferable to C# in some scenarios.

> Because the CLR supports multiple languages, you don't have to pick just one for your whole project. It's common for primarily C#-based

> projects to use C++ to deal with a non-C#-friendly API, using the .NET extensions for C++ (officially called *C++/CLI*) to present a C#-friendly wrapper. The freedom to pick the best tool for the job is useful, but there is a price. The mental 'context switch' developers have to perform when moving between languages takes its toll, and could outweigh the benefits. Mixing languages works best when each language has a very clearly defined role in the project, such as dealing with gnarly APIs.

Of course Windows is not the only platform, and the environment in which your code runs is likely to influence your language choice. Sometimes you will have to target a particular system, e.g., Windows on the desktop, or perhaps iOS on handheld devices, because that's what most of your users happen to be using. But if you're writing a web application, you can choose more or less any server-side language and OS, and still write an application that works just fine for users running any operating system on their desktop, phone or tablet. So even if Windows is ubiquitous on desktops in your organization, you don't necessarily have to use Microsoft's platform on the server. Frankly, there are numerous languages that make it possible to build excellent web applications, so the choice will not come down to language features. It is more likely to be driven by the expertise you have in house. If you have a development shop full of Ruby experts, choosing C# for your next web application might not be the most effective use of the available talent.

So not every project will use C#. But since you've read this far, presumably you're still considering using C#. So what is C# like?

C#'s Defining Features

Although C#'s most superficially obvious feature is its C-family syntax, perhaps its most distinctive feature is that it was the first language designed to be a native in the world of the CLR. As the name suggests, the Common Language Runtime is designed to be flexible enough to support many languages, but there's an important difference between a language that has been extended to support the CLR and one that puts it at the center of its design. The .NET extensions in Microsoft's C++ compiler make this very clear—the syntax for using those features is visibly different from standard C++, making a clear distinction between the native world of C++ and the outside world of the CLR. But even without different syntax[2], there will still be friction when two worlds have different ways of working. For example, if you need a collection of numbers, should you use a standard C++ collection class such as `vector<int>` or one from the .NET Framework such as `List<int>`? Whichever you choose, it will be the wrong one some of the time: C++ libraries won't know what to do with a .NET collection, while .NET APIs won't be able to use the C++ type.

[2] Microsoft's first set of .NET extensions for C++ attempted to resemble ordinary C++ more closely. In the end, it turned out to be less confusing to use a distinct syntax for something that is quite different from ordinary C++, so they deprecated the first system (Managed C++) in favour of the newer, more distinctive syntax, which is called C++/CLI.

C# embraces the .NET Framework, both the runtime and the libraries, so these dilemmas do not arise. In the scenario just discussed, `List<int>` has no rival. There is no friction when using .NET Libraries because they are built for the same world as C#.

That much is also true of Visual Basic, but that language retains links to a pre-.NET world. The .NET version of Visual Basic is in many respects a quite different language than its predecessors, but Microsoft went to some lengths to retain many aspects of older versions. The upshot is that it has several language features that have nothing to do with how the CLR works, and are a veneer that the Visual Basic compiler provides on top of the runtime. There's nothing wrong with that, of course. That's what compilers usually do, and in fact C# has steadily added its own abstractions. But the first version of C# presented a model that was very closely related to the CLR's own model, and the abstractions it has added since have been designed to fit well with the CLR. This gives C# a distinctive feel from other languages.

This means that if you want to understand C#, you need to understand the CLR, and the way in which it runs code. (By the way, I will mainly talk about Microsoft's implementations in this book, but there are specifications that define language and runtime behavior for all C# implementations. See the sidebar, "C#, the CLR, and Standards".)

C#, the CLR, and Standards

The CLR is Microsoft's implementation of the runtime for .NET languages such as C# and Visual Basic. Other implementations such as Mono do not use the CLR, but they have something equivalent. The standards body ECMA has published OS-independent specifications for the various elements required by a C# implementation, and these define more generic names for the various parts. There are two documents: ECMA-334 is the C# Language Specification and ECMA-335 defines the *Common Language Infrastructure* (CLI), the world in which C# programs run. These have also since been published by the International Standards Organization as ISO/IEC 23270:2006 and ISO/IEC 23271:2006. However, as those numbers suggest, these standards are now rather old. They correspond to version 2.0 of .NET and C#. Microsoft has published its own C# specification with each new release, and at the time of writing this, ECMA is working on an updated CLI specification, so be aware that the ratified standards are now some way behind the state of the art. Newer features are still publicly documented, but only in draft form for the CLI.

Version drift notwithstanding, it's not quite accurate to say that the CLR is Microsoft's implementation of the CLI because the scope of the CLI is slightly broader. ECMA-335 defines not just the runtime behavior (which it calls the Virtual Execution System, or VES), but also the file format for executable and library files, the Common Type System (CTS), and a subset of that type system that languages are expected to be able to support to guarantee interoperability between languages, called the Common Language Specification (CLS).

> So you could say that Microsoft's CLI is the entire .NET Framework rather than just the CLR, although .NET includes a lot of additional features not in the CLI specification. (For example, the class library that the CLI demands comprises only a small subset of .NET's much larger library.) The CLR is effectively .NET's VES, but you hardly ever see the term VES used outside of the specification, which is why I mostly talk about the CLR in this book. However, the terms CTS and CLS are more widely used, and I'll refer to them again in this book.
>
> In fact, Microsoft has released more than one implementation of the CLI. The .NET Framework is the commercial quality product, and implements more than just the features of the CLI. They also released a codebase called the Shared Source CLI (SSCLI, also known by its codename, Rotor), which, as the name suggests, is the source code for an implementation of the CLI. This aligns with the latest official standards, so it has not been updated since 2006.

Managed Code and the CLR

For years, the most common way for a compiler to work was to process source code, and to produce output in a form that could be executed directly by the computer's CPU. Compilers would produce *machine code*—a series of instructions in whatever binary format was required by the kind of CPU the computer had. Many compilers still work this way, but the C# compiler does not. Instead, it uses a model called *managed code*.

With managed code, the runtime generates the machine code that the CPU executes, not the compiler. This enables the runtime to provide services that are hard or even impossible to provide under the more traditional model. The compiler produces an intermediate form of binary code, the *Intermediate Language* (IL), and the runtime provides the executable binary at runtime.

Perhaps the most visible benefit of the managed model is that the compiler's output is not tied to a single CPU architecture. You can write a .NET component that can run on the 32-bit x86 architecture that PCs have used for decades, but which will also work well in the newer 64-bit update to that design (x64), and also on completely different architectures such as ARM and Itanium. With a language that compiles directly to machine code, you'd need to build different binaries for each of these. Not only can you compile a single .NET component that can run on any of them, it would even be able to run on platforms that weren't supported at the time you compiled the code, if a suitable runtime becomes available in the future. More generally, any kind of improvement to the CLR's code generation—whether that's support for new CPU architectures, or just performance improvements for existing ones—are instantly of benefit to all .NET languages.

The exact moment at which the CLR generates executable machine code can vary. Typically it uses an approach called *just in time* (JIT) compilation, in which each individual function is compiled at runtime, the first time it runs. However, it doesn't have to work this way. In principle, the CLR could use spare CPU cycles to compile functions it thinks you may use in the future (based on what your program did in the past). Or you can get more aggressive: a program's installer can request machine code generation ahead of time so that the program is compiled before it first runs. Conversely, the CLR can sometimes regenerate code some time after the initial JIT compilation. Diagnostics tools can trigger this, but the CLR could also choose to recompile code to better optimize it for the way the code is being used. Recompilation for optimization is not a documented

feature, but the virtualized nature of managed execution is designed to make such things possible in a way that's invisible to your code. Occasionally, it can make its presence felt. For example, virtualized execution leaves some latitude for when and how the runtime performs certain initialization work, and you can sometimes see the results of its optimizations causing things to happen in a surprising order.

Processor-independent JIT compilation is not the main benefit offered by managed code. The greatest payoff is the set of services the runtime provides. One of the most important of these is memory management. The runtime provides a garbage collector, a service that automatically frees memory that is no longer in use. This means that in most cases, you do not need to write code that explicitly returns memory to the operating system once you have finished using it. Depending on which languages you have used before, either this will be wholly unremarkable, or it will make a profound difference to how you write code.

> Although the garbage collector does take care of most memory handling issues, you can defeat its heuristics, and that sometimes happens by accident. We will look at the GC's operation in more detail in Chapter 7.

Managed code has ubiquitous type information. The file formats dictated by the CLI require this to be present, because it enables certain runtime features. For example, the .NET Framework provides various automatic serialization services, in which objects can be converted into binary or textual representations of their state, and those representations can later be turned back into objects, perhaps on a different machine. This sort of service relies on a complete and accurate description of an object's structure, something that's guaranteed to be present in managed code. Type information can be used in other ways. For example, unit test frameworks can use it to inspect code in a test project and discover all of the unit tests you have written. This relies on the CLR's *reflection* services, which are the topic of Chapter 13.

The availability of type information enables an important security feature. The runtime can check code for type safety, and in certain situations, it will reject code that performs unsafe operations. (One example of unsafe code is the use C-style pointers. Pointer arithmetic can subvert the type system, which in turn can allow you to bypass security mechanisms. C# supports pointers, but the resultant unsafe code that fail the type safety checks.) You can configure .NET to allow only certain code known to be trustworthy to use unsafe features. This makes it possible to support the download and local execution of .NET code from potentially untrustworthy sources (e.g., some random web site) without risk of compromising the user's machine. The Silverlight web browser plugin uses this model by default, because it provides a way to deploy .NET code to a web site that client machines can download and run, and needs to ensure that it does not open up a security hole. It relies on the type information in the code to verify that all the type safety rules are met.

Although C#'s close connection with the runtime is one of its main defining features, it's not the only one. Visual Basic has a similar connection with the CLR, but C# is distinguished from Visual Basic by more than just syntax: it has a somewhat different philosophy.

Generality Trumps Specialization

C# favors general-purpose language features over specialized ones. Over the years, Microsoft has expanded C# several times, and the language's designers always have specific scenarios in mind for new features. However, they try have always tried hard to ensure that each new element they add is useful beyond just the scenario for which it was designed.

For example, one of the goals for C# 3.0 was that database access should feel well integrated with the language. The resulting technology, Language Integrated Query (LINQ), certainly supports that goal, but Microsoft achieved this without adding any direct support for data access to the language. Instead, a series of quite diverse-seeming capabilities were added. These included better support for functional programming idioms, the ability to add new methods to existing types without resorting to inheritance, support for anonymous types, the ability to obtain an object model representing the structure of an expression, and the introduction of query syntax. The last of these has an obvious connection to data access, but the rest are harder to relate to the task at hand. Nonetheless, these can be used collectively in a way that makes certain data access tasks significantly simpler. But the features are all useful in their own right, so as well as supporting data access they enable a much wider range of scenarios. For example, version 3.0 of C# made it very much easier to process lists, sets, and other groups of objects, because the new features work for collections of things from any origin, not just databases.

Perhaps the clearest illustration of this philosophy of generality was a language feature that C# chose not to implement, but which Visual Basic did. In VB, you can write XML directly in your source code, embedding expressions to calculate values for certain bits of content at runtime. This compiles into code that generates the completed XML at runtime. VB also has intrinsic support for queries that extract data from XML documents. These same concepts were considered for C#. Microsoft Research developed extensions for C# supporting embedded XML which were demonstrated publicly some time before the first release of Visual Basic to support this. Nevertheless, this feature didn't ultimately make it into C#. It is a relatively narrow facility, only useful when creating XML documents. As for querying XML documents, C# supports this through its general-purpose LINQ features, and does not add any XML-specific support. XML's star has waned since this language concept was mooted, having been usurped in many cases by JSON (which will doubtless be eclipsed by something else in years to come). Had embedded XML made it into C#, it would by now feel like a slightly anachronistic curiosity.

Having said that, C# 5.0 has a new feature that looks relatively specialized. In fact, it has only one purpose. However, it's an important purpose.

Asynchronous Programming

The most significant new feature in C# 5.0 is support for asynchronous programming. .NET has always offered asynchronous APIs, i.e., ones that do not wait for the operation they perform to finish before returning. Asynchrony is particularly important with I/O operations, which can take a long time and often don't require any active involvement from the CPU except at the start and end of an operation. Simple, synchronous APIs that do not return until the operation completes can be inefficient. They tie up a thread while

waiting, which can cause suboptimal performance in servers, and they're also unhelpful in client-side code where they can make a user interface unresponsive.

The problem with the more efficient and flexible asynchronous APIs has always been that they are considerably harder to use than their synchronous counterparts. But now, if an asynchronous API conforms to a certain pattern, you can write C# code that looks almost as simple as the synchronous alternative would.

Although asynchronous support is a rather specialized aspect of C#, it's still fairly adaptable. It can use the Task Parallel Library (TPL) introduced in .NET 4.0, but the same language feature can also work with the new asynchronous mechanisms that Windows 8 uses in WinRT (the API for writing Metro applications). And if you want to write your own custom asynchronous mechanisms, you can arrange for these to be consumable by the native asynchronous features of the C# language.

I've now described some of the defining features of C#, but Microsoft provides more than just a language and runtime. There's also a development environment that can help you write, test, debug, and maintain your code.

Visual Studio

Visual Studio is Microsoft's development environment. There are various editions ranging from free to eye-wateringly expensive. All versions provide the basic features such as a text editor, build tools, and a debugger, as well as providing visual editing tools for user interfaces. It's not strictly necessary to use Visual Studio—the .NET build system that it uses is available from the command line, so you could use any text editor. But it is the development environment that most C# developers use, so I'll start with a quick introduction to working in Visual Studio.

> You can download the free version of Visual Studio (which Microsoft calls the Express edition) from http://www.microsoft.com/express

Any non-trivial C# project will have multiple source code files, and in Visual Studio these will belong to a *project*. Each project builds a single output, or *target*. The build target might be as simple as a single file—a C# project might produce an executable file or a library[3] for example—but some projects produce more complicated outputs. For instance, some project types build web sites. A web site will normally comprise multiple files, but collectively, these files represent a single entity: one web site. Each project's output will typically be deployed as a unit, even if it consists of multiple files.

Project files usually have extensions ending in *proj*. For example, C# projects have a *.csproj* extension, while C++ projects use *.vcxproj*. If you examine these files with a text editor, you'll find that they usually contain XML. (That's not always true. Visual Studio is extensible, and each type of project is defined by a *project system* that can use whatever format it likes, but the built-in languages use XML.) These files list the contents of the project, and configure how it should be built. The XML format that Visual Studio

[3] Executables typically have a *.exe* file extension in Windows, while libraries use *.dll* (historically short for Dynamic Link Library). These are almost identical, the only difference being that a *.exe* file specifies an application entry point. Both kinds of file can export features to be consumed by other components. These are both examples of assemblies, the subject of Chapter 12.

uses for C# project files can also be processed by the *msbuild* tool, which enables you to build projects from the command line.

You will often want to work with groups of projects. For example, it is good practice to write tests for your code, but most test code does not need to be deployed as part of the application, so we typically put automated tests into separate projects. And we may want to split our code up for other reasons. Perhaps the system you're building has a desktop application and a web site, and you have common code you'd like to use in both applications. In this case, you'd need one project that builds a library containing the common code, another producing the desktop application executable, another to build the web site, and three more projects containing the unit tests for each of the main projects.

Visual Studio helps you to work with multiple related projects through what it calls a *solution*. A solution is simply a collection of projects, and while they are usually related, they don't have to be—a solution is really just a container. You can see the currently loaded solution and all the projects it contains in Visual Studio's *Solution Explorer*. Figure 1-1 shows a solution with two projects. The body of this panel is a tree view, and you can expand each project to see the files that make up that project. This panel is normally open at the top right of Visual Studio, but it's possible to hide or close it. You can re-open it with the View→Solution Explorer menu item.

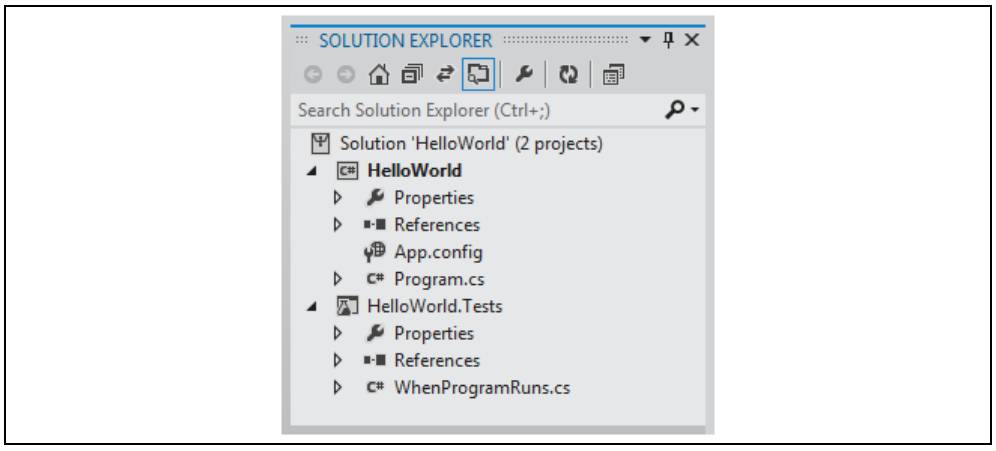

Figure 1-1. Solution Explorer

Visual Studio can only load projects as part of a solution, so when you create a brand new project, you can add it to an existing solution, but if you don't, Visual Studio will create one for you; if you try to open an existing project file, Visual Studio will look for an associated solution, and if it can't find one it will insist that you either provide one, or let it create one. That's because lots of operations in Visual Studio are scoped to the currently-loaded solution. When you build your code, it's normally the solution that you build. Configuration settings, such as a choice between Debug and Release builds, are controlled at the solution level. Global text searches can search all the files in the solution.

A solution is just another text file, with a *.sln* extension. Oddly, it's not an XML file—solution files contain plain text, although also in a format that *msbuild* understands. If you look at the folder containing your solution, you'll also notice a *.suo* file. This is a binary file that contains per-user settings such as a record of which files you have open, and which project or projects to launch when starting debug sessions. That ensures that when

you open a project, everything is more or less where you left it when you last used it. These are per-user settings, so you do not normally check *.suo* files into source control.

A project can belong to more than one solution. In a large codebase, it's common to have multiple *.sln* files with different combinations of projects. You would typically have a master solution that contains every single project, but not all developers will want to work with all the code all of the time. Someone working on the desktop application in our hypothetical example will also want the shared library, but probably has no interest in loading the web project. Larger solutions take longer to load and compile, but they may also require the developer to do extra work—web projects require the developer to have a local web server available, for example. Visual Studio supplies a simple one, but if the project makes use of features specific to a particular server (such as Microsoft's IIS) then you'd need to have that server installed and configured to be able to load the web project. For a developer who was only planning to work on the desktop app, that would be an annoying waste of time. So it would make sense to create a separate solution with just the projects needed for working on the desktop application.

With that in mind, I'll show how to create a new project and solution, and I'll then walk through the various features Visual Studio adds to a new C# project as an introduction to the language. I'll also show how to add a unit test project to the solution.

> This next section is intended for developers who are new to Visual Studio—this book is aimed at experienced developers, but does not assuming any prior experience in C#. The majority of the book is suitable if you have some C# experience and are looking to learn more, but if that's you, you might want to skim through this next section quickly, because you will already be familiar with the development environment by now.

Anatomy of a Simple Program

To create a new project, you can use Visual Studio's File→New→Project menu item, or if you prefer keyboard shortcuts, type Ctrl-Shift-N. This opens the New Project dialog, shown in Figure 1-2. On the left-hand side is a tree view categorizing projects by language, and then project type. I've selected C# of course, and I've chosen the Windows category, which includes not just projects for desktop applications, but also for libraries (DLLs) and console applications. I've selected the latter.

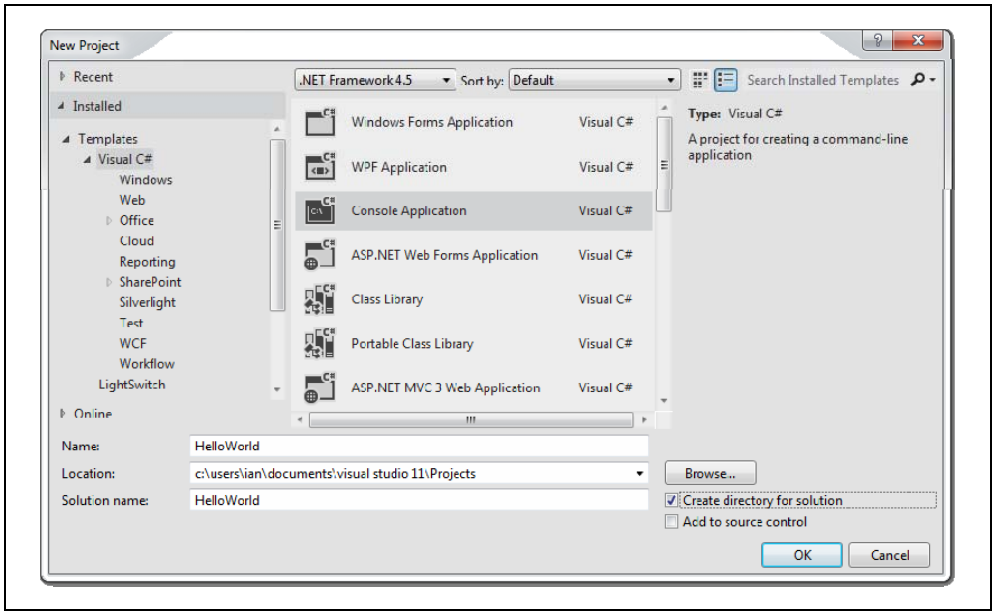

Figure 1-2. The New Project dialog

> Different editions of Visual Studio offer different sets of templates. Also, even within a single edition, the structure of the treeview on the left of the New Project dialog will vary according to the choice you make when you first run Visual Studio. It offers various different configurations according to your language preference. I chose C#, but if you selected something else, C# may be buried one level further down under "Other Languages".

[I will provide a non-ClearType version of this, and all the other VS images in this chapter for the final draft]

Towards the bottom of the dialog, the *Name* field affects three things. It controls the name of the *.csproj* file on disk. It also determines the filename of the compiled output, although you can change that later. Finally, it sets the default namespace for newly-created code, which I'll explain when I show the code. Visual Studio offers a checkbox letting you decide how the associated solution is created. If you set it to unchecked, the project and solution will have the same name and will live in the same folder on disk. But if you plan to add multiple projects to your new solution, you will typically want the solution to be in its own folder, with each project stored in a subfolder. If you check the *Create directory for solution* checkbox, Visual Studio will set things up that way, and it also enables the *Solution name* textbox so you can give the solution a different name from the first project if necessary.

I'm intending to add a unit test project to the solution as well as the program, so I've checked the checkbox. I've set the project name to *HelloWorld*, and Visual Studio has set the solution name to match, which I'm happy with here. Clicking OK creates a new C# project. So I currently have a solution with a single project in it.

Adding a Project to an Existing Solution

To add a unit test project to the solution, I can go to the Solution Explorer panel, right-click on the solution node (the one at the very top) and choose Add→New Project. Alternatively, I can open the New Project dialog again, and this time, because I've already got a solution open, it will offer me an extra choice: I can either add the new project to the current solution, or create a new one.

Apart from that detail, this is the same New Project dialog I used for the first project, and this time, I'll select Visual C#→Test from the categories on the left, and then pick the Unit Test Project project template. This will contain tests for my *HelloWorld* project, so I'll call it *HelloWorld.Tests*. (Nothing demands that naming convention by the way—I could have called it anything.) Clicking OK, Visual Studio creates a second project, and both are now listed in Solution Explorer, which will look similar to Figure 1-1.

The purpose of this test project will be to ensure that the main project does what it's supposed to. I happen to prefer the style of development where you write your tests before you write the code being tested, so we'll start with the test project. (This is sometimes called *Test Driven Development*, or TDD.) To be able to do its job, my test project will need access to the code in the *HelloWorld* project. Visual Studio has no way of guessing which projects in a solution may depend on which other projects. Even though there are only two here, if it tried to guess which depends on the other, it would most likely guess wrong, because *HelloWorld* will produce an *.exe* file, while unit test projects happen to produce a *.dll*. The most obvious guess would be that the *.exe* would depend on the *.dll*, but here we have the somewhat unusual requirement that our library (which is actually a test project) depends on the code in our application.

Referencing One Project from Another

To tell Visual Studio about the relationship between these two projects, we right-click on the *HelloWorld.Test* project's *References* node in Solution Explorer, and select the Add Reference menu item. This shows the Reference Manager dialog, which you can see in Figure 1-3. On the left, you choose the sort of reference you want—in this case, I'm setting up a reference to another project in the same solution, so I have expanded the Solution section and selected Projects. This lists all the other projects in the middle, and there's just one in this case, so I check the *HelloWorld* item and click OK.

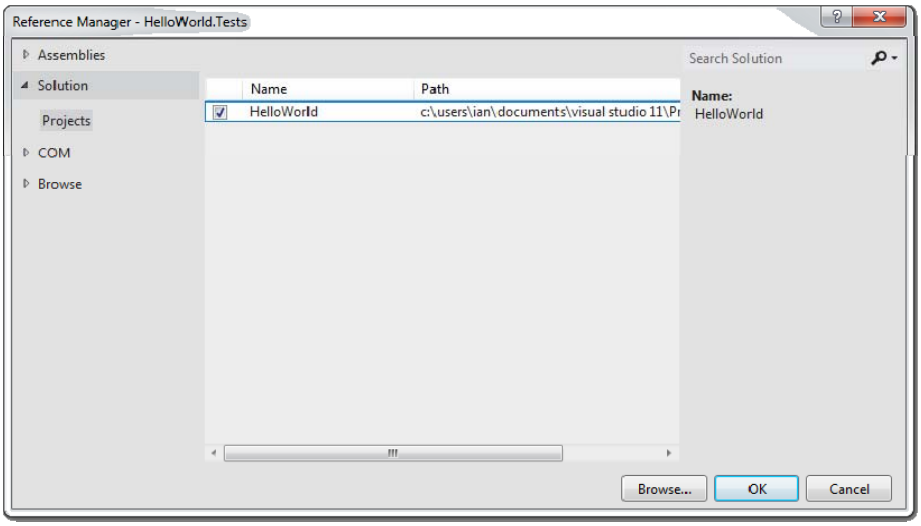

Figure 1-3. The Reference Manager dialog

When you add a reference, Visual Studio expands the References node in Solution Explorer, so that you can see the new reference. As Figure 1-4 shows, this will not be the only reference—a newly created project has references to several standard system components. It does not reference everything in the .NET Framework class library though. Visual Studio will choose the initial set of references based on the project type. Unit test projects get a very small set. More specialized applications such as desktop user interfaces or web applications will get additional references for the relevant parts of the framework. You can always add a reference to any component in the class library using the Reference Manager dialog. If you were to expand the Assemblies section, visible at the top left of Figure 1-3, you'd see two items, Framework and Extensions. The first gives you access to everything in the .NET Framework class library, while the second provides access to other .NET components that have been installed on your machine. (E.g., if you have installed other .NET-based SDKs, their components will appear here.)

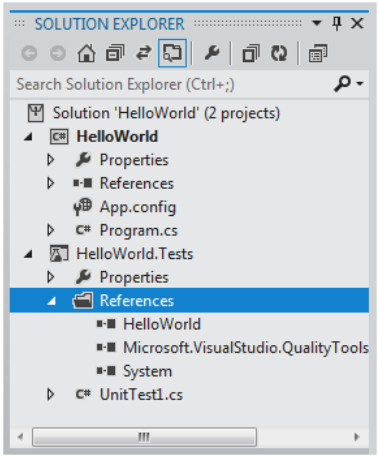

Figure 1-4. References node showing project reference

Writing a Unit Test

Now I need to write a test. Visual Studio has provided me with a test class to get me started in a file called *UnitTest1.cs*, but that's not a very informative name. I need to call it something else. There are various schools of thought as to how you should structure your unit tests. Some developers advocate one test class for each class you wish to test, but I like the style where you write a class for each *scenario* in which you want to test a particular class, with one method for each of the things that should be true about your code in that scenario. As you've probably guessed from the project names I've chosen, our program will only have one behavior: it will display a "Hello, world!" message when it runs. So I'll rename the *UnitTest1.cs* source file to *WhenProgramRuns.cs*. This test should verify that the program prints out the required message when it runs. The test itself is very simple, but unfortunately, getting to the point where we can run the test is a bit more involved. Example 1-1 shows the whole source file; the test is near the end, in bold.

Example 1-1. A unit test class for our first program

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace HelloWorld.Tests
{
    [TestClass]
    public class WhenProgramRuns
    {
        private string _consoleOutput;

        [TestInitialize]
        public void Initialize()
        {
            var w = new System.IO.StringWriter();
            Console.SetOut(w);

            Program.Main(new string[0]);

            _consoleOutput = w.GetStringBuilder().ToString().Trim();
        }

        [TestMethod]
        public void SaysHelloWorld()
        {
            Assert.AreEqual("Hello, world!", _consoleOutput);
        }
    }
}
```

I will explain each of the features in this file once I've shown the program itself. For now, the most interesting part of this is that it defines some behavior we want our program to have. The test states that the program's output should be the message "Hello, world!" If it's not, this test will report a failure. The test itself is pleasingly simple, but the code that sets things up for the test is a little awkward. The problem here is that the obligatory first example that all programming books are required by law to show isn't very amenable to unit testing of individual classes, because you can't really test anything less than the whole program. We want to verify that the program prints out a particular message to the console. In a real application, you'd probably devise some sort of

abstraction for output, and your unit tests would provide a fake version of that abstraction for test purposes. But I want my application (which Example 1-1 merely tests) to keep to the spirit of the standard "Hello, world!" example. To avoid overcomplicating my program, I've made my test intercept console output so that I can check that the program printed what it was supposed to. (Chapter 16 will describe the features I'm using from the `System.IO` namespace to achieve this.)

There's a second challenge. Normally, a unit test will, by definition, test some isolated and usually small part of the program. But in this case, the program is so simple that there is only one feature of interest, and that feature executes when we run the program. This means my test will need to invoke the program's entry point. I could have done that by launching my *HelloWorld* program in a whole new process, but capturing its output would have been rather more complex than the in-process interception done by Example 1-1. Instead, I'm just invoking the program's entry point directly. In a C# application, the entry point is usually a method called `Main` defined in a class called `Program`. Example 1-2 shows the relevant line from Example 1-1, passing an empty array to simulate running the program with no command line arguments.

Example 1-2. Calling a method

```
Program.Main(new string[0]);
```

Unfortunately, there's a problem with that. A program's entry point is typically only accessible to the runtime—it's an implementation detail of your program, and there's not normally any reason to make it publicly accessible. However, I'll make an exception here, because that's where the only code in in this example will live. So to get the code to compile, we'll need to make a change to our main program. Example 1-3 shows the relevant line from the code from the *Program.cs* file in the *HelloWorld* project. (I'll show the whole thing shortly.)

Example 1-3. Making the program entry point accessible

```
public class Program
{
    public static void Main(string[] args)
    {
...
```

I've added the `public` keyword to the start of two lines to make the code accessible to the test, enabling Example 1-1 to compile. There are other ways I could have achieved this. I could have left the class as it is, made the method `internal`, and then applied the `InternalsVisibleToAttribute` to my program to grant access just to the test suite. But internal protection and assembly level attributes are topics for later chapters (3 and 15 respectively) so I decided to keep it simple for this first example. I'll show the alternative approach in Chapter 15.

> Microsoft's unit testing framework defines a helper class called `PrivateType`, which provides a way to invoke private methods for test purposes, and I could have used that instead of making the entry point public. However, it's considered bad practice to invoke private methods directly from tests, because a test should only have to verify the observable behavior of the code under test. Testing specific details of how the code has been structured is rarely helpful.

I'm now ready to run my test. To do this, I open the Unit Test Explorer panel with the Unit Tests→Windows→Unit Test Explorer menu item. Next, I build the project with the Build→Build Solution menu. Once I've done that, the Unit Test explorer shows a list of all the unit tests defined in the solution. It finds my `SayHelloWorld` test as you can see in Figure 1-5. Clicking on Run All runs the test, which, of course, fails, because we've not actually put any code in our main program yet. You can see the error at the bottom of Figure 1-5. It says it was expecting a "Hello, world!" message, but that there was no console output.

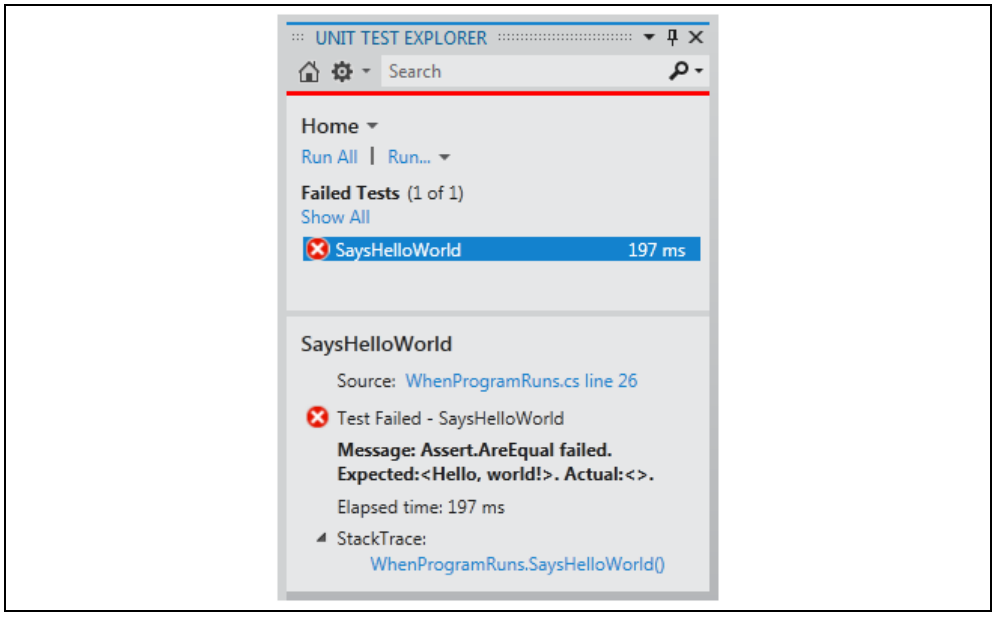

Figure 1-5. Unit Test Explorer

So it's time to look at our *HelloWorld* program, and to add the missing code. When I created the project, Visual Studio generated various files, including *Program.cs*, which contains the program's entry point. Example 1-4 shows this file, including the modifications I made in Example 1-3. I shall explain each element in turn, as it provides a useful introduction to some important elements of C# syntax and structure.

Example 1-4. Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HelloWorld
{
    public class Program
    {
        public static void Main(string[] args)
        {
        }
    }
}
```

The file begins with a series of *using directives*. These are optional, but almost all source files contain them, and they tell the compiler which *namespaces* we'd like to use, raising the obvious question: what's a namespace?

Namespaces

Namespaces bring order and structure to what would otherwise be a horrible mess. The .NET Framework class library contains over 10,000 classes, and there are many more classes out there in 3rd party libraries, not to mention the classes you will write yourself. There are two problems that can occur when dealing with this many named entities. First, it becomes hard to guarantee uniqueness unless everything either has a very long name, or the names include sections of random gibberish. Second, it can become challenging to discover the API you need—unless you know or can guess the right name, it's difficult to find what you need from an unstructured list of thousands of things. Namespaces solve both of these problems.

Most .NET types are defined in a namespace. Microsoft-supplied types have distinctive namespaces. When the types are part of the .NET Framework, the containing namespaces start with `System`, and when they're part of some Microsoft technology which is not a core part of .NET, they usually begin with `Microsoft`. Libraries from other vendors tend to start with the company name, while open source libraries often use their project name. You are not forced to put your own types into namespaces, but it's recommended that you do. C# does not treat `System` as a special namespace, so nothing stops you using that for your own types, but it's a bad idea because it will tend to confuse other developers. You should pick something more distinctive for your own code, such as your company or project name.

The namespace usually gives a clue as to the purpose of the type. For example, all the types that relate to file handling can be found in the `System.IO` namespace, while those concerned with networking are under `System.Net`. Namespaces can form a hierarchy. So the framework's `System` namespace doesn't just contain types. It also holds other namespaces, such as `System.Net`, and these often contain yet more namespaces, such as `System.Net.Sockets` and `System.Net.Mail`. These examples show that namespaces act as a sort of description, which can help you navigate the library. If you were looking for regular expression handling, for example, you might look through the available namespaces, and notice the `System.Text` namespace. Looking in there, you'd find a `System.Text.RegularExpressions` namespace, at which point you'd be pretty confident that you were looking in the right place.

Namespaces also provide a way to ensure uniqueness. The namespace in which a type is defined is part of that type's full name. This lets libraries use short, simple names for things. For example, the regular expression API includes a `Capture` class representing the results from a regular expression capture. If you are working on software that deals with images, the term 'capture' is more commonly used to mean the acquisition of some image data, and you might feel that `Capture` is the most descriptive name for a class in your own code. It would be annoying to have to pick a different name just because the best one is already taken, particularly if your image acquisition code has no use for regular expressions, meaning that you weren't even planning to use that `Capture` type.

But in fact it's fine. Both types can be called `Capture`, and they will still have different names. The full name of the regular expression capture class is effectively `System.Text.RegularExpressions.Capture`, and likewise your class's full

name would include its containing namespace, e.g. `SpiffingSoftworks.Imaging.Capture`.

If you really want to, you can write the fully qualified name of a type every time you use it, but most developers don't want to do anything quite so tedious, which is where the `using` directives at the start of Example 1-4 come in. These state the namespaces whose types this source file intends to use. You will normally edit this list to match your file's requirements, but Visual Studio provides a small selection of commonly used ones to get you started. It chooses different sets in different contexts. If you add a class representing a user interface control, for example, Visual Studio would include various UI-related namespaces in the list.

With `using` declarations like these in place, you can just use the short, unqualified name for a class. When we finally add the line of code that enables our *HelloWorld* example to do its job, we'll be using the `System.Console` class, but because of the first `using` directive, we'll be able to refer to it as just `Console`. In fact, that's the only class we'll be using, so we could remove all the other `using` directives.

> Earlier, you saw that a project's References describe which libraries it uses. You might think that References are redundant—can't the compiler work out which external libraries we are using from the namespaces? It could if there were a direct correspondence between namespaces and libraries, but there isn't. There is sometimes an apparent connection—*System.Web.dll* contains classes in the `System.Web` namespace for example. But there often isn't—the class library includes *System.Core.dll* but there is no `System.Core` namespace. So it is necessary to tell Visual Studio which libraries your project depends on, as well as saying which namespaces any particular source file uses. We will look at the nature and structure of library files in more detail in Chapter 12.

Even with namespaces, there's potential for ambiguity. You might use two namespaces that both happen to define a class of the same name. If you want to use that class, then you will need to be explicit, referring to it by its full name. If you need to use such classes a lot in the file, you can still save yourself some typing: you only need to use the full name once because you can define an *alias*. Example 1-5 uses this to resolve a clash that I've run into a few times: .NET's user interface framework, the Windows Presentation Foundation (WPF) defines a `Path` class for working with Bézier curves, polygons and other shapes, but there's also a `Path` class for working with filesystem paths, and if you want to write code that produces a graphical representation of the contents of a file, you will need both namespaces, meaning that the simple name `Path` is ambiguous if unqualified. But as Example 1-5 shows, you can define distinctive aliases for each.

Example 1-5. Resolving ambiguity with aliases

```
using System.IO;
using System.Windows.Shapes;

using IoPath = System.IO.Path;
using WpfPath = System.Windows.Shapes.Path;
```

With these aliases in place, you can use `IoPath` as a synonym for the file-related `Path` class, and `WpfPath` for the graphical one.

Going back to our *HelloWorld* example, directly after the `using` directives comes a *namespace declaration*. Whereas `using` directives declare which namespaces our code will consume, a namespace declaration declares the namespace in which our own code lives. Example 1-6 shows the relevant code from Example 1-4. This is followed by an opening brace ({). Everything between this and the closing brace at the end of the file will be in the `HelloWorld` namespace. By the way, you can refer to types in your own namespace without qualification, without needing a `using` directive.

Example 1-6. Namespace declaration

```
namespace HelloWorld
{
```

Visual Studio generates a namespace declaration with the same name as your project. You're not required to keep this—a project can contain any mixture of namespaces, and you are free to edit the namespace declaration. But if you do want to use something other than the project name consistently throughout your project, you should tell Visual Studio because it's not just the first file, *Program.cs*, that gets this generated declaration. By default, Visual Studio adds a namespace declaration based on your project name every time you add a new file. You can tell it to use a different namespace for new files by editing the project's properties. If you double click on the *Properties* node inside a project in Solution Explorer, this opens the properties for the project, and if you go to the *Application* tab there's a *Default namespace* textbox. It will use whatever you put in there for namespace declarations of any new files. (It won't change the existing files though.)

Nested namespaces

The .NET Framework class library nests namespaces, and sometimes quite extensively. The `System` namespace contains numerous important types, but most types are in more specific namespaces such as `System.Net`, or `System.Net.Sockets`. If the complexity of your project demands it, you can also nest your own namespaces. There are two ways you can do this. You can nest namespace declarations, as Example 1-7 shows.

Example 1-7. Nesting namespace declarations

```
namespace MyApp
{
    namespace Storage
    {
        ...
    }
}
```

Alternatively, you can just specify the full namespace in a single declaration, as Example 1-8 shows. This is the more commonly used style.

Example 1-8. Nested namespace with a single declaration

```
namespace MyApp.Storage
{
    ...
}
```

Any code you write in a nested namespace will be able to use types not just from that namespace, but also its containing namespaces without qualification. Code in either Example 1-7 or Example 1-8 would be not need either explicit qualification or `using`

directives to use types either in the `MyApp.Storage` namespace, or the `MyApp` namespace.

When you define nested namespaces, the convention is to define a folder for each namespace. If you create a project called `MyApp`, by default Visual Studio will put new classes in the `MyApp` namespace when you add them to the project, and if you create a new folder in the project (which you can do in Solution Explorer) called, say, `Storage`, Visual Studio will put any new classes you create in that folder into the `MyApp.Storage` namespace. Again, you're not required to keep this—Visual Studio just adds a namespace declaration when creating the file, and you're free to change it. The compiler does not care if the namespace does not match your folder hierarchy. But since the convention is supported by Visual Studio, life will be easier if you follow it.

Classes

Inside the namespace declaration, our *Program.cs* file defines a *class*. Example 1-9 shows this part of the file (which includes the `public` keywords I added earlier). The `class` keyword is followed by the name, and of course the full name of the type is effectively `HelloWorld.Program`, because this code is inside the namespace declaration. As you can see, C# uses braces ({}) to delimit all sorts of things—we already saw this for namespaces, and here you can see the same thing with the class and also the method it contains.

Example 1-9. A class with a method

```
public class Program
{
    public static void Main(string[] args)
    {
    }
}
```

Classes are C#'s mechanism for defining entities that combine state and behavior, a common object-oriented idiom. But as it happens, this class contains nothing more than a single method. C# does not support global methods—all code has to be written as a member of some type. So this particular class isn't very interesting—its only job is to act as the container for the program's entry point. We'll see some more interesting uses for classes in Chapter 3.

Program Entry Point

By default, the C# compiler will look for a method called `Main`, and use that as the entry point automatically. If you really want to you can tell the compiler to use a different method, but most programs stick with the convention. Whether you designate the entry point by configuration or convention, the method has to meet certain requirements, all of which are evident in Example 1-9.

The program entry point must be a *static method*, meaning that it is not necessary to create an instance of the containing type (`Program` in this case) in order to invoke the method. It is not required to return anything, as signified by the `void` keyword here, although if you want to you can return `int` instead, which allows the program to return an exit code that the operating system will report when the program terminates. And the method must either take no arguments at all (which would be denoted by an empty pair of

parentheses after the method name) or, as in Example 1-9, the method can accept a single argument: an array of text strings containing the command line arguments.

> Some C family languages include the filename of the program itself as the first argument, on the grounds that it's part of what the user typed at the command prompt. C# does not follow this convention. If the program is launched without arguments, the array's length will be zero.

The method declaration is followed by the method body which is currently empty. We've now looked at everything that Visual Studio generated for us in this file, so all that remains is to add some code inside the braces delimiting the method body. Remember, our test is failing, because our program fails to meet its one requirement: to print out a certain message to the console. This requires the single line of code shown in Example 1-10, placed inside the method body.

Example 1-10. Printing a message

```
Console.WriteLine("Hello, world!");
```

With this in place, if I run the tests again, the unit test explorer shows a tick by my test and reports that all tests have passed. So apparently the code is working. And we can verify that informally by running the program. You can do that from Visual Studio's Debug menu. The Start Debugging option runs the program in the debugger, although you'll find it runs so quickly that it finishes before you have a chance to see the output. So you might want to choose Start Without Debugging—this runs without attaching the Visual Studio debugger, but it also runs the program in such a way as to leave the console window that shows the program's output visible after the program finishes. So if you run the program this way (which you can also do with the Ctrl-F5 keyboard shortcut) you'll see it display the traditional message in a window that stays open until you press a key.

Unit Tests

Now that our program is working, I want to go back to the first code I wrote, the test, because that file illustrates some C# features that our main program does not. If you go back to Example 1-1, it starts in a pretty similar way to the main program: we have a series of `using` directives, and then a namespace declaration, the namespace being `HelloWorld.Tests` this time, matching the test project name. But the class looks different. Example 1-11 shows the relevant part of Example 1-1.

Example 1-11. Test class with attribute

```
[TestClass]
public class WhenProgramRuns
{
```

Immediately before the class declaration is the text `[TestClass]`. This is an *attribute*. Attributes are annotations you can apply to classes, methods, and other features of the code. Most of them do nothing on their own—the compiler records the fact that the attribute is present in the compiled output, but that is all. Attributes are only useful when something goes looking for them, so they tend to be used by frameworks. In this case, I'm using Microsoft's unit testing framework, and it goes looking for classes annotated with this `TestClass` attribute. It will ignore classes that do not have this annotation. Attributes are typically specific to a particular framework, and you can define your own, as we'll see in Chapter 15.

The two methods in the class are also annotated with attributes. Example 1-9 shows the relevant excerpts from Example 1-1. The test runner will execute any methods marked with `[TestInitialize]` once for every test the class contains, and does so before running the actual test method itself. And as you have no doubt guessed, the `[TestMethod]` attribute tells the test runner which methods represent tests.

Example 1-12. Annotated methods

```
[TestInitialize]
public void Initialize()
...

[TestMethod]
public void SaysHelloWorld()
...
```

There's one more feature in Example 1-1: the class contents begin with a field, shown in Example 1-13. Fields hold data. In this case, the `Initialize` method stores the console output that it captures while the program runs in this `_consoleOutput` field, where it is available for test methods to inspect. This particular field has been marked as `private`, indicating that it is for this particular class's own use. The C# compiler will only permit code that lives in the same class to access this data.

Example 1-13. A field

```
private string _consoleOutput;
```

And with that, we've examined every element of a program, and the test project that verifies that it works as intended.

Summary

You've now seen the basic structure of C# programs. I created a Visual Studio solution containing two projects, one for tests, and one for the program itself. This was a simple example, so each project only had one source file of interest. Both were of similar structure. Each began with `using` directives indicating which types the file uses. A namespace declaration stated the namespace that the file populates, and this contained a class containing one or more methods or other members such as fields.

We will look at types and their members in much more detail in Chapter 3, but first, Chapter 2 will deal with the code that lives inside methods, where we express what we want our programs to do.

2

Basic Coding in C#

All programming languages have to provide certain capabilities. It must be possible to express the calculations and operations that our code should perform. Programs need to be able to make decisions based on their input. Sometimes we will need to perform tasks repeatedly. These fundamental features are the very stuff of programming, and this chapter will show how these things work in C#.

Depending on your background, some of this chapter's content may seem very familiar. C# is said to be from the "C family" of languages. C is a hugely influential programming language, and numerous languages have borrowed much of its syntax. There are direct descendants such as C++ and Objective C. There are also more distantly related languages, including Java and JavaScript, that have no compatibility with, but still ape many aspects of C's syntax. If you are familiar with any of these languages, you will recognize most of the basic language features we are about to explore.

We saw the basic structure of a program in Chapter 1. In this chapter, I will be looking just at code inside methods. C# requires a certain amount of structure: code is made up of statements that live inside a method, which belongs to a type, which is typically inside a namespace, all inside a file that is part of a Visual Studio project. For clarity, most of the examples in this chapter will show the code of interest in isolation, as in Example 2-1.

Example 2-1. The code, and nothing but the code

```
Console.WriteLine("Hello, world!");
```

Unless I say otherwise, a short extract like that is shorthand for showing the code in context inside a suitable program. So Example 2-1 is short for Example 2-2.

Example 2-2. The whole code

```
using System;

namespace Hello
{
    class Program
    {
        static void Main()
        {
```

```
            Console.WriteLine("Hello, world!");
        }
    }
}
```

Although I'll be introducing fundamental elements of the language in this section, this book is for people who are already familiar with at least one programming language, so I'll be relatively brief with the most ordinary aspects of the language, and will go into most detail on the aspects peculiar to C#.

Local Variables

The inevitable "Hello, world!" example is missing a pretty crucial feature as programs go: it doesn't really deal with information. Useful programs normally fetch, process, and produce information, so the ability to define and identify information is one of the most important features of a language. Like most languages, C# lets you define *local variables*, which are named elements inside a method that each hold a piece of information.

> In the C# specification, the term *variable* can refer to local variables, but also to fields in objects, and array elements. This section is concerned entirely with local variables, but it gets tiring to keep reading the 'local' prefix. So from now on in this section 'variable' means a local variable.

C# is a *statically-typed* language which is to say that any element of code that represents or produces information, such as a variable, or an expression, has its data type determined at compile time. This is different than *dynamically-typed* languages such as JavaScript, in which types are determined at runtime.[1]

The easiest way to see C#'s static typing in action is with simple variable declarations such as the ones in Example 2-3. Each of these starts with the data type—the first two variables are of type `string`, and the next two are `int`.

Example 2-3. Variable declarations

```
string part1 = "the ultimate question";
string part2 = "of something";
int theAnswer = 42;
int something;
```

The data type is followed immediately by the variable's name. The name must begin with either a letter or an underscore, which can be followed by any combination of the characters described in the 'Identifier and Pattern Syntax' annex of the Unicode specification. If you're just using text in the ASCII range, that means letters, decimal digits, and underscores. If you're using Unicode's full range, this also includes various accents, diacritics, and numerous somewhat obscure punctuation marks. These same rules determine what constitutes a legal identifier for any user-defined entity in C#, such as a class or a method.

[1] C# does in fact offer dynamic typing as an option with its `dynamic` keyword, but it takes the slightly unusual step of fitting that into a statically-typed point of view: dynamic variables have a static type of `dynamic`. Chapter 14 will explain all that.

Example 2-3 shows that there are a couple of forms of variable declaration. The first three variables include an *initializer*, providing the variable's initial value, but as the final variable shows, this is optional. That's because you can assign new values into variables at any point. Example 2-4 continues on from Example 2-3, and shows that you can assign a new value into a variable regardless of whether it had an initial value.

Example 2-4. Assigning values to previously declared variables

```
part2 = " of life, the universe, and everything";
something = 123;
```

Because variables have a static type, the compiler will reject attempts to assign the wrong kind of data. So if we were to follow on from Example 2-3 with the code in Example 2-5, the compiler would complain. It knows that the `theAnswer` variable's type is `int`, which is a numeric type, so it will report an error when we attempt to assign a text string into it.

Example 2-5. An error: the wrong type

```
theAnswer = "The compiler will reject this";
```

You'd be allowed to do this in a dynamic language such as JavaScript, because in those languages, a variable's type is defined by whatever's in that variable at any particular moment. Not so in C#, unless you declare a variable as `dynamic`, which I'll get to in Chapter 14.

> The static type doesn't always provide a complete picture, thanks to inheritance. I'll be discussing this in Chapter 6, but for now, it's enough to know that some types are open to extension through inheritance, and if a variable uses such a type, then it's possible for it to refer to some object of a type derived from the variable's static type. Interfaces, described in Chapter 3, provide a similar kind of flexibility. However, the static type always determines what operations you are allowed to perform on the variable. If you want to use additional features specific to some derived type, you won't be able to do so through a variable of the base type.

You don't have to state the variable type explicitly. You can let the compiler work it out for you by using the keyword `var` in place of the data type. Example 2-6 shows the first three variable declarations from Example 2-3, but using `var` instead of explicit data types.

Example 2-6. Implicit variable types with the var keyword

```
var part1 = "the ultimate question";
var part2 = "of something";
var theAnswer = 42;
```

This code often misleads people who know some JavaScript, because that also has a `var` keyword that you can use in a similar-looking way. But `var` does not work the same way in C# as in JavaScript: these variables are still all statically typed. All that's changed is that we haven't said what the type is—we're letting the compiler deduce it for us. It looks at the initializers, and can see that the first two variables are strings while the third is an integer. (That's why I left out the fourth variable from Example 2-3, `something`. That doesn't have an initializer, so the compiler would have no way of inferring its type. If you try to use the `var` keyword without an initializer, you'll get a compiler error.)

You can demonstrate that variables declared with `var` are statically typed by attempting to assign something of a different type into them. We could repeat the same thing we tried in Example 2-5, but this time with a `var`-style variable. Example 2-7 does this, and it will produce exactly the same compiler error, because it's the same mistake—we're trying to assign a text string into a variable of an incompatible type. That variable, `theAnswer`, has a type of `int` here, even though we didn't say so explicitly.

Example 2-7. An error: the wrong type (again)

```
var theAnswer = 42;
theAnswer = "The compiler will reject this";
```

To var, or not to var?

A `var`-style variable declaration is exactly equivalent to a variable declaration with an explicit type, which raises a question: which should you use? In a sense, it doesn't matter, because they are equivalent. However if you like your code to be consistent, you'll probably want to pick one style and stick to it. Opinion varies as to which is the 'best' style.

Some developers dislike expending more keystrokes than they absolutely have to. They may refer contemptuously to the extra text required for explicit variable types as unproductive 'ceremony' that should be replaced with the more succinct `var` keyword. The compiler can work out the type for you, so you should let it do the work instead of doing it yourself, or so the argument goes.

Your author takes a different view. I find that I spend more time reading my code than I did writing it—activities such as debugging, refactoring, or modifying the functionality seem to dominate. Anything that makes those activities easier is worth the frankly minimal time it takes to write the type names explicitly. Code that uses `var` everywhere slows you down, because you have to work out what the type really is in order to understand the code. Although the compiler saved you some work when you wrote the code, that gain is quickly wiped out by the additional thought required every time you go back and look at the code. So unless you're the sort of developer who only ever writes new code, leaving others to clean up after you, the `var` everywhere philosophy seems to have little to commend it.

That said, there are some situations in which I will use `var`. One is in code where explicit typing would mean writing the name of the type twice. For example, if you initialize a variable with a new object, you could write this:

```
List<int> numbers = new List<int>();
```

In this case, there's no downside to using `var`, because the type name is right there in the initializer, so you won't need to expend any mental effort to work out the type when reading the implicit version:

```
var numbers = new List<int>();
```

There are similar examples involving casts, and generic methods; the principle here is that as long as the type name appears explicitly in the variable declaration, it's OK to use `var` to avoid writing the type twice.

> The other situations in which I use `var` are where it is necessary. As we shall see in later chapters, C# supports *anonymous types*, and as the name suggests, it's not actually possible to write the name of such a type. In these situations, you may be compelled to use `var`. (In fact, the `var` keyword was only introduced to C# when anonymous types were added.)

One last thing worth knowing about declarations is that you can declare, and optionally initialize, multiple variables in a single line. If you want multiple variables of the same type, this may reduce clutter in your code. Example 2-8 declares three variables of the same type in a single declaration.

Example 2-8. Multiple variables in a single declaration

```
double a = 1, b = 2.5, c = -3;
```

In summary, a variable holds some piece of information of a particular type, and the compiler prevents us from putting data of an incompatible type into that variable. Of course, variables are only useful because we can refer back to them later in our code. Example 2-9 starts with the variable declarations we saw in earlier examples, and then goes on to use the values of those variables to initialize some more variables, and then prints out the results.

Example 2-9. Using variables

```
string part1 = "the ultimate question";
string part2 = "of something";
int theAnswer = 42;

part2 = "of life, the universe, and everything";

string questionText = "What is the answer to " + part1 + ", " + part2 + "?";
string answerText = "The answer to " + part1 + ", " +
                        part2 + ", is: " + theAnswer;

Console.WriteLine(questionText);
Console.WriteLine(answerText);
```

By the way, this code relies on the fact that C# defines a couple of meanings for the `+` operator when used with strings. When you 'add' two strings together, it concatenates them. When you 'add' a number to the end of a string (as the initializer for `answerText` does), C# generates code that converts the number to a string before appending it. So Example 2-9 produces this output:

```
What is the answer to the ultimate question, of life, the universe, and everythi
ng?
The answer to the ultimate question, of life, the universe, and everything, is:
42
```

> In this book, text longer than 80 columns is wrapped across multiple lines to fit. If you try these examples, they will look different if your console windows are configured for a different width.

When you use a variable, its value is whatever you last assigned into it. If you attempt to use a variable before you have assigned a value, as Example 2-10 does, the C# compiler will report an error.

Example 2-10. Error: using an unassigned variable

```
int willNotWork;
Console.WriteLine(willNotWork);
```

Compiling that produces this error for the second line:

```
error CS0165: Use of unassigned local variable 'willNotWork'
```

The compiler uses a slightly pessimistic system (which it calls the *definite assignment* rules) for determining whether a variable has a value yet. It's not possible to create an algorithm that can determine such things for certain in every possible situation.[2] Since the compiler has to err on the side of caution, there are some situations in which the variable will have a value by the time the offending code runs, and yet the compiler still complains. The solution is to write an initializer, so that the variable always contains something. For an unused initial value, you'd typically use 0 for numeric values, and `false` for Boolean variables. In Chapter 3, I'll introduce reference types, and as the name suggests, a variable of such a type can hold a reference to an instance of the type. If you need to initialize such a variable before you've got something for it to refer to, you can use the keyword `null`, a special value signifying a reference to nothing.

The definite assignment rules determine the parts of your code in which the compiler considers a variable to contain a valid value, and will therefore let you read from it. Writing into a variable is less restricted, but of course, any given variable is only accessible from certain parts of the code. Let's look at the rules that govern this.

Scope

A variable's *scope* is the range of code in which you can refer to that variable by its name. Local variables are not the only things with scope. Methods, properties, types, and in fact anything with a name all have scope. These require a slightly broader definition of scope: it's the region in which you can refer to the entity by its name without needing additional qualification. When I write `Console.WriteLine` I am referring to the method by its name (`WriteLine`), but I need to qualify it with a class name (`Console`) because the method is not in scope. But with a local variable, scope is absolute: either it's accessible without qualification or it's not accessible at all.

Broadly speaking, a local variable's scope starts at its declaration, and finishes at the end of its containing *block*. A block is a region of code delimited by a pair of braces ({}). A method body is a block, so a variable defined in one method is not visible in another method, because it is out of scope. If you attempt to compile Example 2-11, you'll get an error complaining that "`The name 'thisWillNotWork' does not exist in the current context`".

Example 2-11. Error: out of scope

```
static void SomeMethod()
{
    int thisWillNotWork = 42;
}

static void AnotherMethod()
```

[2] See Alan Turing's seminal work on computation for details. Charles Petzold's "The Annotated Turing" from John Wiley & Sons is an excellent guide to the relevant paper.

```
{
    Console.WriteLine(thisWillNotWork);
}
```

Methods often contain nested blocks, particularly when you work with the loop and flow control constructs we'll be looking at later in this chapter. At the point where a nested block starts, everything that is in scope in the outer block continues to be in scope inside that nested block. Example 2-12 declares a variable called `someValue`, and then introduces a nested block as part of an `if` statement. The code inside this block is able to access that variable declared in the containing block.

Example 2-12. Variable declared outside block, used within block

```
int someValue = GetValue();
if (someValue > 100)
{
    Console.WriteLine(someValue);
}
```

The converse is not true. If you declare a variable in a nested block, its scope does not extend outside of that block. So Example 2-13 will fail to compile, because the `willNotWork` variable is only in scope within the nested block. The final line of code will produce a compiler error because it's trying to use that variable outside of that block.

Example 2-13. Error: trying to use a variable not in scope

```
int someValue = GetValue();
if (someValue > 100)
{
    int willNotWork = someValue - 100;
}
Console.WriteLine(willNotWork);
```

This probably all seems fairly straightforward, but things get a bit more complex when it comes to potential naming collisions. C# sometimes catches people by surprise here.

Variable name ambiguity

Consider the code in Example 2-14. This declares a variable called `anotherValue` inside a nested block. As you know, that variable is only in scope to the end of that nested block. After that block ends, we try to declare another variable with the same name.

Example 2-14. Error: surprising name collision overlap

```
int someValue = GetValue();
if (someValue > 100)
{
    int anotherValue = someValue - 100;
    Console.WriteLine(anotherValue);
}

int anotherValue = 123;
```

This causes a compiler error on the final line:

```
error CS0136: A local variable named 'anotherValue' cannot be declared in this
scope because it would give a different meaning to 'anotherValue', which is already
used in a 'child' scope to denote something else
```

This seems odd. At the final line, the supposedly conflicting earlier declaration is not in scope, because we're outside of the nested block in which it was declared. Furthermore, the second declaration is not in scope within that nested block, because the declaration comes after the block. The scopes do not overlap, but despite this, we've fallen foul of C#'s rules for avoiding name conflicts. To see why this example fails, we first need to look at a less surprising example.

C# tries to prevent ambiguity by disallowing code where one name might refer to more than one thing. Example 2-15 shows the sort of problem it aims to avoid. Here we've got a variable called `errorCount`, and the code starts to modify this as it progresses, but part way through, it introduces a new variable in a nested block, also called `errorCount`. It would be possible to imagine a language that allowed this—you could have a rule that says that when multiple items of the same name are in scope, you just pick the one whose declaration happened last.

Example 2-15. Error: hiding a variable

```
int errorCount = 0;
if (problem1)
{
    errorCount += 1;

    if (problem2)
    {
        errorCount += 1;
    }

    int errorCount = GetErrors();  // Compiler error
    if (problem3)
    {
        errorCount += 1;
    }
}
```

In fact, the compiler does not allow this, because code that did this would be easy to misunderstand. This is an artificially short method because it's a fake example in a book, so the problem is clear. If the code were a bit longer, it would be very easy to miss the nested variable declaration, and not to realize that `errorCount` refers to something different at the end of the method than it did earlier on. C# simply disallows this to avoid misunderstanding.

But why does Example 2-14 fail? The scopes of the two variables don't overlap. Well, it turns out that the rule that outlaws Example 2-15 is not based on scopes. It is based on a subtly different concept called a *declaration space*. A declaration space is a region of code in which a single name must not refer to two different entities. Each method defines a declaration space for variables. Nested blocks also introduce declaration spaces, and it is illegal for a nested declaration space to declare a variable with the same name as one in its parent's declaration space. And that's the rule we've fallen foul of here—the outermost declaration space in Example 2-15 contains a variable named `errorCount`, and a nested block's declaration space tries to introduce another variable of the same name.

If that all seems a bit dry, it may be helpful to know *why* there's a whole separate set of rules for name collisions instead of basing it on scopes. The intent of the declaration space rules is that it mostly shouldn't matter where you put the declaration. If you were to

move all of the variable declarations in a block to the start of that block—and some organizations have coding standards that mandate this sort of layout—the idea of these rules is that this shouldn't change what the code means. Clearly this wouldn't be possible if Example 2-15 were legal. And this explains why Example 2-14 is illegal. Although the scopes don't overlap, they would if you moved all variable declarations to the top of their containing blocks.

Local variable instances

A variable is a feature of the source code, so a particular variable has a distinct identity: it is declared in exactly one place in the source code, and goes out of scope at exactly one well-defined place. However, that doesn't mean that it corresponds to a single storage location in memory. It is possible for multiple invocations of a single method to be in progress simultaneously, either through recursion or multithreading.

Each time a method runs, it gets a separate set of storage locations to hold the values corresponding to the local variables for that run. So in multi-threaded code, threads will not interfere with each other when working with variables. Likewise, in recursive code, each nested call gets its own set of locals that will not interfere with any of its callers.

Be aware that the C# compiler does not make any particular guarantee about where local variables live. They might well live on the stack, but they don't have to. When we look at anonymous methods in later chapters, you'll see that local variables sometimes need to outlive the method that declares them, because they remain in scope for nested methods that will run as callbacks in the future.

By the way, before we move on, be aware that just as variables are not the only things to have scope, they are also not the only things to which declaration space rules apply. Other language features that we'll be looking at later, including classes, methods, and properties, also have scoping and name uniqueness rules.

Statements and Expressions

Variables let us define the information that our code works with, but to do anything with those variables, we will need to write some code. This will mean writing *statements* and *expressions*.

Statements

When we write a C# method, we are writing a sequence of statements. Informally, the statements in a method describe the actions we want the method to perform. Each line in Example 2-16 is a statement. It might be tempting to think of a statement as an instruction to do one thing (e.g., initialize a variable, or invoke a method). Or you might take a more lexical view, where anything ending in a semicolon is a statement. However, both descriptions are simplistic, even though they happen to be true for this particular example.

Example 2-16. Some statements

```
int a = 19;
int b = 23;
int c;
c = a + b;
Console.WriteLine(c);
```

C# recognizes many different kinds of statements. The first three lines of Example 2-16 are *declaration statements*, statements which declare and optionally initialize a variable. The fourth and fifth lines are *expression statements* (and we'll be looking at expressions shortly). But some statements have more structure than the ones in this example.

When you write a loop, that's an *iteration statement*. When you use the `if` or `select` mechanisms described later in this chapter to choose between various possible actions, those are *selection statements*. In fact the C# specification distinguishes between 14 different categories of statement. Most fit broadly into the scheme of describing either what the code should do next, or, for features such as loops or conditional statements, describing *how* it should do what it does next. Statements of that second kind usually contain one or more embedded statements describing the action to perform in a loop, or the action to perform when an `if` statement's condition is met.

There's one special case though. A block is a kind of statement. This makes statements like loops more useful than they would otherwise be because a loop iterates over just a single embedded statement. That statement can be a block, and since a block itself is a sequence of statements (delimited by curly brackets: {}), this is what enables loops to contain more than one statement.

This illustrates why the two simplistic points of view stated earlier—"expressions are actions" and "expressions are things that end in semicolons"—are wrong. Compare Example 2-16 with Example 2-17. Both do the same thing because the various actions we've said we want to perform remain exactly the same. However, Example 2-17 contains one extra statement. The first two statements are the same, but they are followed by a third statement, a block, which contains the final three statements from Example 2-16. The extra statement, the block, doesn't end in a semicolon, nor does it perform any action. It might seem pointless, but it can sometimes be useful to introduce a nested block like this to avoid name ambiguity errors. So statements can be structural, rather than causing anything to happen at runtime.

Example 2-17. A block

```
int a = 19;
int b = 23;
{
    int c;
    c = a + b;
    Console.WriteLine(c);
}
```

While your code will contain a mixture of statement types, it will inevitably end up containing at least a few expression statements. These are, quite simply, statements that consist of a suitable expression, followed by a semicolon. What's a suitable expression? What's an expression for that matter? I'd better answer that before coming back to what constitutes a valid expression for a statement.

Expressions

The official definition of a C# expression is rather dry: "a sequence of operators and operands." Admittedly, language specifications tend to be like that, but in addition to this sort of formal prose, the C# specification contains some very readable informal explanations of the more formally expressed ideas. (It describes statements as the means by which "the actions of a program are expressed" for example, before going on to pin

that down with less approachable but more technically precise language.) I'm quoting from the formal definition of an expression at the start of this paragraph, so perhaps the informal explanation in the introduction will be more helpful. That part says that expressions "are constructed from operands and operators." That's certainly less precise than the other definition, but no easier to understand. The problem is that there are several kinds of expression, and they do different jobs, so there isn't a single, general, informal description.

It's tempting to describe an expression as some code that produces a value. That's not true for all expressions, but the majority of expressions you'll write will fit this description, so I'll focus on this for now, and I'll come to the exceptions later.

The simplest expressions with values are *literals*, where we just write the value we want, such as `"Hello, world!"` or `2`. You can also use the name of a variable as an expression. Expressions can also involve operators, which describe calculations or other computations to be performed. Operators have some fixed number of inputs, or *operands*. Some take a single operand. For example, you can negate a number by putting a minus sign in front of it. Some take two: the `+` operator lets you form an expression which adds together the results of the two operands on either side of the `+` symbol.

> Some symbols have different roles depending on the context. The minus sign is not just used for negation. It acts as a two-operand subtraction operator if it appears between two expressions.

In general, operands are also expressions. So when we write `2 + 2`, that's an expression that contains two more expressions, the pair of '`2`' literals on either side of the `+` symbol. This means that we can write arbitrarily complicated expressions by nesting expressions within expressions within expressions. Example 2-18 exploits this to evaluate the quadratic formula (the standard technique for solving quadratic equations).

Example 2-18. Expressions within expressions

```
double a = 1, b = 2.5, c = -3;
double x = (-b + Math.Sqrt(b * b - 4 * a * c)) / (2 * a);
Console.WriteLine(x);
```

Look at the declaration statement on the second line. Its initializer expression's overall structure is a division operation. But that division operator's two operands are also expressions. Its left hand operand is a *parenthesized expression*, which tells the compiler that I want that whole expression `(-b + Math.Sqrt(b * b - 4 * a * c))` to be the first operand. This subexpression contains an addition, whose left hand operand is a negation expression whose single operand is the variable `b`. The addition's right hand side takes the square root of another, more complex expression. And the division's left-hand operand is another parenthesized expression, containing a multiplication. Figure 2-1 illustrates the full structure of the expression.

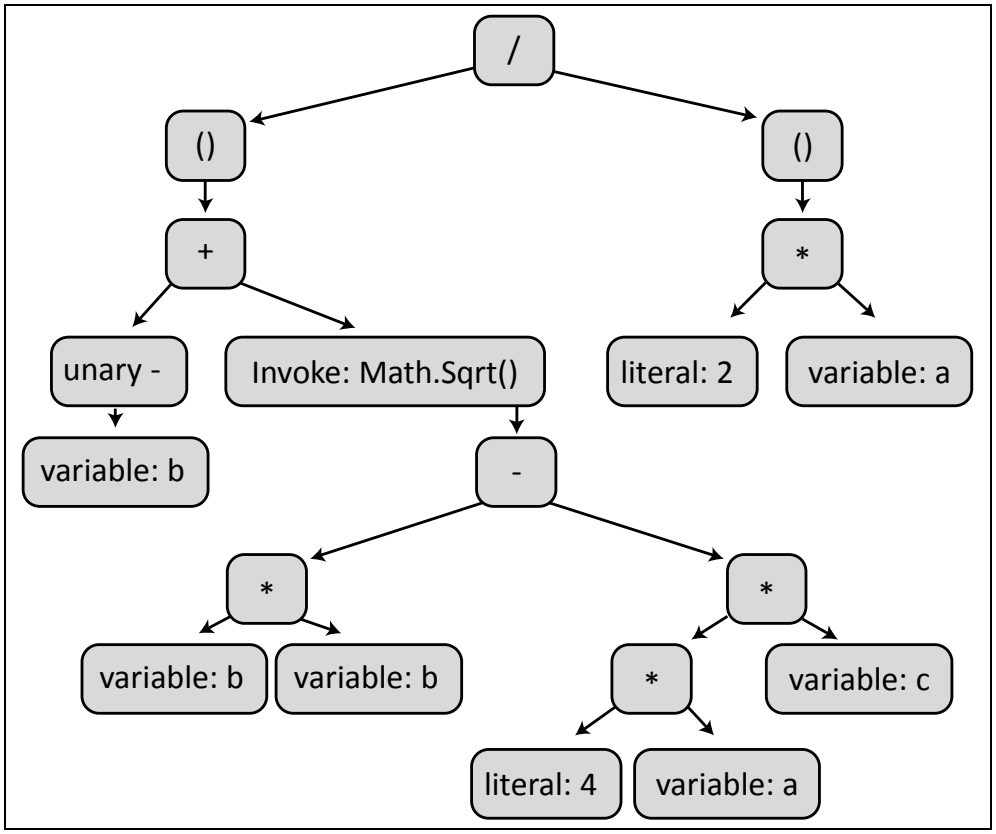

Figure 2-1. The structure of an expression

One important detail of this last example is that method invocations are a kind of expression. The `Math.Sqrt` method used in Example 2-18 is a .NET Framework class library function that calculates the square root of its input and returns the result. What's perhaps more surprising is that invocations of methods that don't return a value, such as `Console.WriteLine`, are also, technically, expressions. And there are a few other constructs that don't produce values but which are still considered to be expressions, including a reference to a type (e.g. the `Console` in `Console.WriteLine`) or to a namespace. These sorts of constructs take advantage of a set of common rules (e.g., scoping, how to resolve what a name refers to, etc.). However, all the non-value-producing expressions can only be used in certain specific circumstances. (You can't use one as an operand in another expression, for example.) So although it's not technically correct to define an expression as a piece of code that produces a value, the ones that do are the ones we use when describing the calculations we want our code to perform.

So we can now return to the question: what can we put in an expression statement? Roughly speaking, the expression has to do something; it cannot just calculate a value. So although `2 + 2` is a valid expression, you'll get an error if you try to turn it into an expression statement by sticking a semicolon on the end. That expression calculates something but doesn't do anything with the result. To be more precise, you can use the following kinds of expressions as statements: method invocation, assignment, increment, decrement, and new object creation. We'll be looking at increment and decrement later in

this chapter, and we'll be looking at objects in later chapters, so that leaves invocation and assignment.

So a method invocation is allowed to be an expression statement. This can involve nested expressions of other kinds, but the whole thing must be a method call. Example 2-19 shows some valid examples. Notice that the C# compiler doesn't check whether the method call really has any lasting effect—the `Math.Sqrt` function is a pure function, in the sense that it does nothing other than returning a value based entirely on its inputs. So invoking it and then doing nothing with the result doesn't really do anything at all—it's no more of an action than the expression `2 + 2`. But as far as the C# compiler is concerned, any method call is allowed as an expression statement.

Example 2-19. Method invocation expressions as statements

```
Console.WriteLine("Hello, world!");
Console.WriteLine(12 + 30);
Console.ReadKey();
Math.Sqrt(4);
```

It seems inconsistent that C# forbids us from using an addition expression as a statement while allowing `Math.Sqrt`. Both attempt to perform a calculation and then discard the result? Wouldn't it be more consistent if C# only allowed calls to methods that return nothing to be used for expression statements? That would rule out the final line of Example 2-19, which would seem like a good idea because that code does nothing useful. However, sometimes you want to ignore the return value. Example 2-19 calls `Console.ReadKey()`, which waits for a keypress and returns a value indicating which key was pressed. If my program's behavior depends on which particular key the user pressed, I'll need to inspect the method's return value, but if I just want to wait for any key at all, it's OK to ignore the return value. If C# didn't allow methods with return values to be used as expression statements, I wouldn't be able to do this. The compiler doesn't know which methods make for pointless statements because they have no side effects such as `Math.Sqrt`, and which might make sense, such as `Console.ReadKey`, so it allows any method.

For an expression to be a valid expression statement, it is not enough merely to contain a method invocation. Example 2-20 shows some expressions that call methods and then go on to use those as part of addition expressions. So these will cause compiler errors.

Example 2-20. Errors: some expressions that don't work as statements

```
Console.ReadKey().KeyChar + "!";
Math.Sqrt(4) + 1;
```

Earlier I said that one of the kinds of expression we're allowed to use as a statement is an assignment. It's not obvious that assignments should be expressions, but they are, and they do produce a value: the result of an assignment expression is the value being assigned into the variable. This means it's legal to write code like that in Example 2-21. The second line here uses an assignment expression as an argument for a method invocation which prints out the value of that expression. The code prints out 123 twice.

Example 2-21. Assignments are expressions

```
int number;
Console.WriteLine(number = 123);
Console.WriteLine(number);
```

This shows that evaluating an expression can do more than just producing a value. Some expressions have side effects. We've just seen that an assignment is an expression, and it of course has the effect of changing what's in a variable. Method calls are expressions too, and although you can write pure functions that do nothing besides calculating their result from their input, like `Math.Sqrt`, many methods do something with lasting effects, such as printing data to the console, updating a database, or launching a missile. This means that we might care about the order in which the operands of an expression get evaluated.

An expression's structure imposes some constraints on the order in which operators do their work. For example, I can use parentheses to enforce ordering. The expression `10 + (8 / 2)` has the value 14, while the expression `(10 + 8) / 2` has the value 9, even though both have exactly the same literal operands and arithmetic operators. The parentheses here determine whether the division is performed before or after the subtraction.[3] However, this is separate from the question of the order in which the operands are evaluated. For these simple expressions, it doesn't matter because I've used literals, so we can't really tell when they get evaluated. But what about an expression in which operands call some method? Example 2-22 contains code of this kind.

Example 2-22. Operand evaluation order

```
class Program
{
    static int X(string label, int i)
    {
        Console.Write(label);
        return i;
    }

    static void Main(string[] args)
    {
        Console.WriteLine(X("a", 1) + X("b", 1) + X("c", 1) + X("d", 1));
        Console.WriteLine();
        Console.WriteLine(
            X("a", 1) +
            X("b", (X("c", 1) + X("d", 1) + X("e", 1))) +
            X("f", 1));
    }
}
```

This defines a method, `X`, which takes two arguments. It prints out the first, and just returns the second. I've then used this in a couple of expressions, and it lets us see exactly when the operands that call `X` are evaluated. Some languages choose not to define this order, making the behavior of such a program unpredictable, but C# does in fact specify an order here. The rule is that within any expression, the operands are evaluated from left to right. So for the first `Console.WriteLine` in Example 2-22, we see it print `abcd4`. Nested expressions complicate things a little, although the same rule applies. The final `Console.WriteLine` adds the results of three calls to `X`, however, the

[3] In the absence of parentheses, C# has rules of *precedence* that determine the order in which operators are evaluated. For the full (and not very interesting) details, consult the C# specification, but in this case, division has higher precedence than addition, so without parentheses, the expression would evaluate to 14.

second of those calls to `X` takes as its argument an expression that adds the results of three more calls to `X`. Starting at the top level additions, the first operand, `X("a", 1)`, will be evaluated. Then it will start to evaluate the second operand, which is that second method call expression. The same rule applies for this subexpression: it will evaluate its operands—the method arguments in this case—from left to right. The first is the constant `"b"`, and the second is the nested expression containing three further calls to `X`, which will also be evaluated from left to right. Once it has evaluated those, it can complete the call to `X` for which that result was the second operand—the call with a first argument of `"b"`. And once that's done, the left-to-right evaluation of the top level of additions continues with the final argument. So we see output of `acdebf5`. So looking at the expression as a whole, the various method calls were not evaluated in the order in which they were written, but that's because they were at various different levels of nesting. Taking any single expression in isolation, it evaluated its operands from left to right, and it's only because those operands are expressions in their own right that we see nested ordering.

Comments and Whitespace

Most programming languages allow source files to contain text that is ignored by the compiler, and C# is no exception. As with most C family languages, it supports two styles of *comments* for this purpose. There are *single-line comments*, in which you write two `/` characters in a row, and everything from there to the end of the line will be ignored by the compiler.

Example 2-23. Single line comments

```
Console.WriteLine("Say");        // This text will be ignored but the code on
Console.WriteLine("Anything");   // the left is still compiled as usual.
```

C# also supports *delimited comments*. These can span multiple lines—you start a comment of this kind with `/*` and the compiler will ignore everything that follows until it encounters the first `*/` character sequence. In Example 2-24, a comment begins in the middle of the first line, and ends at the end of the fourth.

Example 2-24. Multiline comments

```
Console.WriteLine("This will run");   /* This comment includes not just the
Console.WriteLine("This won't");       * text on the right, but also the text
Console.WriteLine("Nor will this");   /* on the left except the first and last
Console.WriteLine("Nor this");         * lines. */
Console.WriteLine("This will also run");
```

Notice that the `/*` character sequence appears twice in this example. When this sequence appears in the middle of a comment, it does nothing special—comments don't nest. Even though we've seen two `/*` sequences, the first `*/` is enough to end the comment. This is occasionally frustrating, but it's the norm for C family languages.

Occasionally, it's useful to take a block of code out of action temporarily, in a way that's easy to put back. Turning the code into a comment is an easy way to do this, but a delimited comment is a bad way to do it, because if the region you commented out happens to include a delimited comment, you won't be able to comment out anything beyond its closing `*/` without starting another comment. So we normally use the single-line comment for this purpose.

> Visual Studio can comment out regions of code for you. If you select several lines of text, and type Ctrl-K followed immediately by Ctrl-C, it will add `//` to the start of every line in the selection. And you can uncomment a region with Ctrl-K, Ctrl-U. If you chose something other than C# as your preferred language when you first ran Visual Studio, these actions may be bound to different key sequences but they are also available on the Edit→Advanced menu, and also on the Text Editor toolbar, one of the standard toolbars that Visual Studio shows by default.

Speaking of ignored text, for the most part C# ignores extra whitespace. Not all whitespace is insignificant, because you need at least some space to separate tokens that consist entirely of alphanumeric symbols. For example, you can't write `staticvoid` as the start of a method declaration—you'd need at least one space (or tab, newline, or other space-like character) between `static` and `void`. But with non-alphanumeric tokens, spaces are optional, and in most cases, a single space is equivalent to any amount of whitespace and new lines. This means that the three statements in Example 2-25 are all equivalent.

Example 2-25. Insignificant whitespace

```
Console.WriteLine("Testing");
Console . WriteLine("Testing");
Console.
    WriteLine("Testing")
  ;
```

There a couple of cases where C# is more sensitive to whitespace. Inside a string literal, space is significant, because whatever spaces you write will be present in the string value. Also, while C# mostly doesn't care whether you put each element on its own line, or you put all your code in one massive line or, as seems more likely, something in between, there is an exception: preprocessing directives are required to appear on their own lines.

Preprocessing Directives

If you're familiar with the C language or its direct descendants, you may have been wondering if C# has a preprocessor. It doesn't have a separate preprocessing stage, and does not offer macros. However, it does have a handful of directives similar to those offered by the C preprocessor, although it is only a very limited selection.

Compilation Symbols

C# offers a `#define` directive which lets you define a *compilation symbol*. These symbols are commonly used to compile code in different ways for different situations. For example, you might want some code to be present only in debug builds, or perhaps you need to use different code on different platforms to achieve a particular effect. Often, you won't use the `#define` directive though—it's more common to define compilation symbols through the compiler build settings. Visual Studio lets you configure different symbol values for each build configuration. To control this, double click the project's Properties node in Solution Explorer, and in the property page that this opens, go to the Build tab. If you're running the compiler from the command line, there are switches to set such things.

> Visual Studio sets certain symbols by default in newly created projects. It will typically create two configurations, Debug and Release. It defines a `DEBUG` compilation symbol in the Debug configuration but not in the Release configuration. It defines a symbol called `TRACE` in both Debug and Release builds. Certain project types get addition symbols. Silverlight projects will have a `SILVERLIGHT` symbol defined in all configurations, for example.

Compilation symbols are typically used in conjunction with the `#if`, `#else`, `#elif`, and `#endif` directives. Example 2-26 uses some of these directives to ensure that certain lines of code only get compiled in debug builds.

Example 2-26. Conditional compilation

```
#if DEBUG
    Console.WriteLine("Starting work");
#endif
    DoWork();
#if DEBUG
    Console.WriteLine("Finished work");
#endif
```

C# provides a more subtle mechanism to support this sort of thing, called a *conditional method*. The compiler recognizes an attribute defined by the .NET Framework, called `ConditionalAttribute`, for which it provides special compile-time behaviors. You can annotate any method with this attribute. Example 2-27 uses it to indicate that the annotated method should only be used when the `DEBUG` compilation symbol is defined.

Example 2-27. Conditional method

```
[System.Diagnostics.Conditional("DEBUG")]
static void ShowDebugInfo(object o)
{
    Console.WriteLine(o);
}
```

If you call a method that has been annotated in this way, the C# compiler will effectively remove the code that makes that call in builds that do not have the relevant symbol defined. So if you write code that calls this `ShowDebugInfo` method, the compiler strips out all those calls in non-debug builds. So you can get the same effect as Example 2-26, without cluttering up your code with directives.

The .NET Framework's `Debug` and `Trace` classes in the `System.Diagnostics` namespace uses this feature. The `Debug` class offers various methods that are conditional on the `DEBUG` compilation symbol, while the `Trace` class has methods conditional on `TRACE`. If you leave the default settings for a new Visual Studio project in place, any diagnostic output produced through the `Trace` class will be available in both Debug and Release builds, but any code that calls a method on the `Debug` class will not get compiled into Release builds.

> The `Debug` class's `Assert` method is also conditional on `DEBUG`. `Assert` lets you specify a condition that must be true at runtime, and it throws an exception if the condition is false. There are two things developers new to C# often mistakenly put in a `Debug.Assert`:

checks that should in fact occur in all builds, and expressions with side effects that the rest of the code depends on. This leads to bugs, because the compiler will strip this code out in non-Debug builds.

#error and #warning

C# lets you choose to generate compiler errors or warnings with the `#error` and `#warning` directives. These are typically used inside of conditional regions, as Example 2-28 shows, although an unconditional `#warning` could be useful as a way to remind yourself that you've not written some particularly important bit of the code yet.

Example 2-28. Generating a compiler error

```
#if SILVERLIGHT
  #error Silverlight is not a supported platform for this source file
#endif
```

#line

The `#line` directive is useful in generated code. When the compiler produces an error or a warning, it normally states where the problem occurred, providing the filename, a line number, and an offset within that line. But if the code in question was generated automatically using some other file as input, and if that other file contains the root cause of the problem, it may be more useful to report an error in the input file, rather than the generated file. A `#line` directive can instruct the C# compiler to act as though the error occurred at the line number specified, and optionally, as if the error were in an entirely different file. Example 2-29 shows how to use it. The error after the directive will be reported as though it came from line 123 of a file called *Foo.cs*.

Example 2-29. The #line directive and a deliberate mistake

```
#line 123 "Foo.cs"
    intt x;
```

The filename part is optional, enabling you to fake just line numbers. You can tell the compiler to revert to reporting warnings and errors without fakery by writing `#line default`.

There's another use for this directive. Instead of a line number (and optional file name) you can instead write just `#line hidden`. This only affects the debugger behavior: when single stepping, Visual Studio will run straight through all the code after such a directive without stopping until it encounters a non-`hidden` `#line` directive.

#pragma

The `#pragma` directive allows you to disable selected compiler warnings. The reason for the slightly idiosyncratic name is that it's modeled on more general compiler control mechanisms found in other C-like languages. And it's possible that future versions of C# may add other features based on this directive. (In fact when the compiler encounters a pragma it does not understand, it generates a warning, not an error, on the grounds that an unrecognized pragma might be valid for some future compiler version, or some other vendor's compiler.)

Example 2-30 shows how to use a `#pragma` to prevent the compiler from issuing the warning that would normally occur if you declare a variable that you do not then go on to use.

Example 2-30. Disabling a compiler warning

```
#pragma warning disable 168
    int a;
```

You should generally avoid disabling warnings. The main use for this feature is in generated code scenarios. Code generation can often end up creating items which are not used, and pragmas may offer the only way to get a clean compilation. But when you're writing code by hand, it should usually be possible to avoid warnings in the first place.

#region and #endregion

Finally, we have two preprocessing directives that do nothing. If you write `#region` directives, the only thing the compiler does is ensure that they have corresponding `#endregion` directives. Mismatches cause compiler errors, but the compiler ignores correctly paired `#region` and `#endregion` directives.

These directives exist entirely for the benefit of text editors that choose to recognize them. Visual Studio uses them to provide the ability to collapse sections of the code down to a single line on screen. The C# editor automatically allows certain features to be expanded and collapsed, such as methods, and class definitions, but if you define regions with these two directives, it will also allow those to be expanded and collapsed. Some people find this useful, as by collapsing all the regions, you can see a file's structure at a glance. It may all fit on the screen at once, thanks to the regions being reduced to a single line. On the other hand, some people hate collapsed regions, because they present speed bumps on the way to being able to look at the code.

Intrinsic Data Types

The .NET Framework defines thousands of types in its class library, and you can write your own, so C# can work with an unlimited number of data types. However, a handful of data types get special treatment from the compiler. You saw earlier in Example 2-9 that if you have a string, and you try to add a number to it, the compiler will generate code that converts the number to a string and appends it to the first string. In fact, the behavior is more general than that—it's not limited to numbers. If you have a string, and you add to it some value of any type that's not a string, the compiler just calls the `ToString` method on whatever you're trying to add, and then calls the `String.Concat` method to combine the string with the result. All types offer a `ToString` method, so this means you can append values of any type to a string.

That's handy, but it only works because the C# compiler knows about strings, and provides special services for them. (There's a part of the C# specification that defines this special string handling for the `+` operator.) C# provides various special services not just for strings, but also certain numeric data types, Booleans, and a type called `object`.

Numeric Types

C# supports integer and floating point arithmetic. There are both signed and unsigned versions of the integer types, and they come in various sizes, as Table 2-1 shows. The most commonly used integer type is `int`, not least because it is large enough to represent a usefully wide range of values, without being too large to work efficiently on all CPUs that support .NET. (Larger data types might not be handled natively by the CPU, and can also have undesirable characteristics in multithreaded code: reads and writes are atomic for 32-bit types[4], but may not be for larger ones.)

Table 2-1. Integer types

C# Type	CLR Name	Signed	Size in bits	Inclusive Range
`byte`	`System.Byte`	No	8	0-255
`sbyte`	`System.SByte`	Yes	8	−128 - 127
`ushort`	`System.UInt16`	No	16	0 - 65535
`short`	`System.Int16`	Yes	16	−32768 - 32767
`uint`	`System.UInt32`	No	32	0-4294967295
`int`	`System.Int32`	Yes	32	−2147483648 - 2147483647
`ulong`	`System.UInt64`	No	64	0 - 18446744073709551615
`long`	`System.Int64`	Yes	64	−9223372036854775808 - 9223372036854775807

The second column in Table 2-1 shows the name of the type in the CLR. Different languages have different naming conventions, and C# uses names from its C family roots for numeric types. But those don't fit with the naming conventions that .NET has for its data types. So as far as the runtime is concerned the names in the second column are the real names—there are various APIs that can report information about types at runtime, and they report these CLR names, not the C# ones. The names are synonymous in C# source code, so you're free to use the runtime names if you want to, but the C# names are arguably a better stylistic fit—keywords in C family languages are all lower case. Since the compiler handles these types differently than most, it's arguably good to have them stand out.

> Not all .NET languages support unsigned numbers, so the .NET Framework class library tends to avoid them, as should you if you are writing a library designed for use from multiple languages. A runtime such as the CLR that supports multiple languages faces a tradeoff between offering a type system rich enough to cover most languages' needs, and forcing an overcomplicated type system on simple languages. To resolve this, .NET's type system, the CTS, is reasonably

[4] Strictly speaking, this is only guaranteed for correctly-aligned 32-bit types. However, C# aligns them correctly by default, and you'd normally only encounter misaligned data in interop scenarios, which are discussed in chapter 22.

> comprehensive, but languages don't have to support all of it. The Common Language Specification (CLS) identifies a relatively small subset of the CTS that all languages should support. Signed integers are in the CLS but unsigned ones are not. You can use non-CLS types freely in your private implementation details but you should limit your public API to CLS types if you want to interoperate with other languages.

C# also supports floating point numbers. There are two types: `float` and `double`, which are 32-bit and 64-bit numbers in the standard IEEE 754 formats, and as the CLR names in Table 2-2 suggest, these correspond to what are commonly called single precision and double precision numbers. Floating point values do not work in the same way as integers, so the range is expressed differently in this table. It shows the smallest non-zero values and the largest values that can be represented. (These can be either positive or negative.)

Table 2-2. Floating point types

C# Type	CLR Name	Size in bits	Precision	Range (Magnitude)
`Float`	`System.Single`	32	23 bits (~7 decimal digits)	1.5×10^{-45} to 3.4×10^{38}
`Double`	`System.Double`	64	52 bits (~15 decimal digits)	5.0×10^{-324} to 1.7×10^{308}

There's a third numeric representation that C# recognizes, called `decimal` (or `System.Decimal` in .NET). This is a 128-bit value, so it can offer greater precision than the other formats, but it is designed for calculations that require predictable handling of decimal fractions. Neither `float` nor `double` can offer that. If you write code that initializes a variable of type `float` to 0 and then adds 0.1 to it 9 times in a row, you might expect to get a value of 0.9, but in fact you'll get 0.9000001. That's because IEEE 754 floating point stores numbers in binary, which cannot represent all decimal fractions. Some are fine—the decimal 0.5 works fine in binary—written in base 2 it's 0.1. But the decimal 0.1 turns into a recurring number in binary. (Specifically, it's 0.0 followed by the recurring sequence 0011.) This means `float` and `double` can only represent an approximation of 0.1, and more generally, only a few decimals can be represented completely accurately. This isn't always instantly obvious, because when floating point numbers are converted to text, they are rounded to a decimal approximation that can mask the discrepancy. But over multiple calculations, the inaccuracies tend to add up, and eventually produce surprising looking results.

For some kinds of calculations this doesn't really matter—in simulations or signal processing, for example, some noise and error is expected. But accountants tend to be less forgiving—little discrepancies like this can make it look like money has magically vanished or appeared. We need calculations that involve money to be absolutely accurate, which makes floating point a terrible choice for such work. So C# also offers the `decimal` type, which is able to offer a well-defined level of decimal precision.

A `decimal` stores numbers as a sign bit (positive or negative) and a pair of integers. There's a 96-bit integer, and the value of the decimal is this first integer (negated if the sign bit says so) multiplied by 10 raised to the power of the second integer which is a number in the range of 0 to −28.[5] 96 bits is enough to represent any 28 digit decimal integer (and some, but not all 29-digit numbers). That's why the second integer, the one representing the power of 10 by which the first is multiplied, has to be between zero and minus 28: it effectively says where the decimal point goes. This format makes it possible to represent any decimal with 28 or fewer digits accurately.

When you write a literal numeric value, you can choose the type. If you write a plain integer such as `123`, its type will be either `int`, `uint`, `long`, or `ulong`—the compiler picks the first type from that list with a range that contains the value. (So `123` would be `int`, `3000000000` would be `uint`, `5000000000` would be `long` etc.) If you write a number with a decimal point such as `1.23`, its type is `double`.

You can tell the compiler that you want a specific type by adding a suffix. So `123U` is a `uint`, `123L` is a `long`, and `123UL` is a `ulong`. Suffix letters are case- and order-independent, so instead of `123UL`, you could instead write `123Lu`, or `123uL`, or any other permutation. For `double`, `float`, and `decimal`, use the `D`, `F`, and `M` suffixes respectively.

These last three types all support a decimal exponential literal format for large numbers, where you put the letter `E` in the constant followed by the power. For example, the literal value `1.5E-20` is the value 1.5 multiplied by 10^{-20}. (This happens to be of type `double`, because that's the default for a number with a decimal point, regardless of whether it's in exponential format. You could write `1.5E-20F` and `1.5E-20M` for `float` and `decimal` constants with equivalent values.)

It's often useful to be able to write integer literals in hexadecimal, because the digits map better onto the binary representation used at runtime. This is particularly important when different bit ranges of a number represent different things. For example, you may need to deal with a numeric return code from a COM component. These codes use the topmost bit to indicate success or failure, and the next few bits to indicate the origin of the error, and the remaining bits to identify the specific error. For example, the COM error code, E_ACCESSDENIED has the value -2,147,024,891. It's hard to see the structure in decimal, but in hexadecimal, it's easier: 80070005. The 007 part indicates that this was originally a plain Win32 error that has been translated into a COM error, and then the remaining bits indicate that the Win32 error code was 5 (ERROR_ACCESS_DENIED). C# lets you write integer literals in hexadecimal for scenarios like these, where the hex representation is more readable. You just prefix the number with `0x`, so in this case you would write `0x80070005`.

Numeric conversions

Each of the built-in numeric types uses a different representation for storing numbers in memory. Converting from one form to another requires some work—even the number 1

[5] 24 of a decimal's 128 bits are therefore unused. Making it smaller would cause alignment difficulties, and using the additional bits for extra precision would have a significant performance impact, because integers whose length is a multiple of 32 bits are easier for a 32-bit CPU to deal with than the alternatives.

looks quite different if you inspect its binary representations as a `float`, an `int`, and a `decimal`. However, C# is able to generate code that converts between formats, and it will often do so automatically. Example 2-31 shows some scenarios in which this will happen.

Example 2-31. Implicit conversions

```
int i = 42;
double di = i;
Console.WriteLine(i / 5);
Console.WriteLine(di / 5);
Console.WriteLine(i / 5.0);
```

The second line assigns the value of an `int` variable into a `double` variable. The C# compiler will generate the necessary code to convert the integer value into its equivalent (or nearest approximately equivalent) floating point value. More subtly, the last two lines will perform similar conversions, as we can see from the output of that code:

```
8
8.4
8.4
```

This shows that the first division produced an integer result—dividing the integer variable `i` by the integer literal 5 caused the compiler to generate code that performs integer division, so the result is 8. But the other two divisions produced a floating point result. In the second case, we've divided the `double` variable `di` by an integer literal 5. C# converts that 5 to floating point before performing the division. And in the final line, we're dividing an integer variable by a floating point literal. This time, it's the variable's value that gets turned from an integer into a floating point value before the division takes place.

In general, when you perform arithmetic calculations that involve a mixture of numeric types, C# will pick the type with the largest range, and *promote* values of types with a narrower range into that larger one before performing the calculations. (Arithmetic operators generally require all their operands to be the same type, so one type has to 'win' for any particular operator.) For example, `double` can represent any value that `int` can, and many that it cannot, so `double` is the more expressive type.[6]

C# will perform numeric conversions implicitly whenever the conversion is a promotion (i.e., the target type has a wider range than the source) because there is no possibility of the conversion failing. However, it will not implicitly convert in the other direction. The second and third lines of Example 2-32 will fail to compile, because they attempt to assign expressions of type `double` into an `int`, which is a *narrowing* conversion, meaning that the source might contain values that are out of the target's range.

Example 2-32. Errors: implicit conversions not available

```
int i = 42;
int willFail = 42.0;
int willAlsoFail = i / 1.0;
```

[6] Promotions are not in fact a feature of C#. C# has a more general mechanism: conversion operators. Promotions are built on top of this—C# defines intrinsic implicit conversion operators for the built-in data types. The promotions discussed here occur as a result of the compiler following its usual rules for conversions.

It is possible to convert in this direction, just not implicitly. You can use a *cast*, where you specify the name of the type to which you'd like to convert in parentheses. Example 2-33 shows a modified version of Example 2-32, where we've stated explicitly that we want a conversion to `int`, and we either don't mind that this conversion might not work correctly, or we have reason to believe that in this specific case the value will be in range. Note that a cast applies just to the first expression that follows it, rather the whole expression, so I've had to use parentheses on the final line. That makes the cast apply to the parenthesized expression; otherwise, it would apply just to the `i` variable, and since that's already an `int`, it would have no effect.

Example 2-33. Explicit conversions with casts

```
int i = 42;
int i2 = (int) 42.0;
int i3 = (int) (i / 1.0);
```

So narrowing conversions require explicit casts, and conversions that cannot lose information occur implicitly. However, with some combinations of types, neither is strictly more expressive than the other. What should happen if you try to add an `int` to a `uint`? Or an `int` to a `float`? These types are all 32 bits in size, so none of them can possibly offer more than 2^{32} distinct values, but they have different ranges, which means that each has values it can represent that the other types cannot. For example, you can represent the value 3,000,000,001 in a `uint`, but it's too large for an `int`, and can only be approximated in a `float`. As floating point numbers get larger, the values that can be represented get further apart—a `float` can represent 3,000,000,000 and also 3,000,001,024, but nothing in between. So for the value 3,000,000,001, `uint` seems better than `float`. But what about −1? That's a negative number, so `uint` can't cope with that. Then there are very large numbers that `float` can represent that are out of range for both `int` and `uint`. Each of these types has its strengths and weaknesses, and it makes no sense to say that in general, one of them is better than the rest.

Perhaps surprisingly, C# allows some implicit conversions even in these potentially lossy scenarios. It only cares about range, not precision: implicit conversions are allowed if the target type has a wider range than the source type. So you can convert from either `int` or `uint` to `float`, because although `float` is unable to represent some values exactly, there are no `int` or `uint` values that it cannot at least approximate. But implicit conversions are not allowed in the other direction because there are some values that are simply too big—unlike `float`, the integer types can't offer approximations for bigger numbers.

You might be wondering what happens if you force a narrowing conversion to `int` with a cast, as Example 2-33 does, in situations where the number is out of range. The answer depends on the type from which you are casting. Conversion from one integer type to another works differently than conversion from floating point to integer. In fact, the C# specification does not define what you get when floating point numbers that are too big get cast to an integer type—the result could be anything. But when casting between integer types, the binary representation is simply reinterpreted. If the two types are of different sizes, the binary will be either truncated or padded to make it the right size for the target type, and then the bits are just treated as if they are of the target type. This is occasionally useful, but can more often produce surprising results, so you can choose an alternative behavior for any out-of-range cast by making it a *checked* conversion.

Checked contexts

C# defines the `checked` keyword, which you can put in front of either a statement or an expression, making it a *checked context*. This means that certain arithmetic operations, including casts, are checked for range overflow at runtime. If you cast a value to an integer type in a checked context, and the value is too high or low to fit, an error will occur—the code will throw a `System.OverflowException`.

As well as checking casts, a checked context will also detect range overflows in ordinary arithmetic. Addition, subtraction, and other operations can take a value beyond the range of its data type. For integers, this typically causes the value to 'roll over', so adding 1 to the maximum value produces the minimum value, and vice versa for subtraction. Occasionally, this wrapping can be useful. For example, if you want to determine how much time has elapsed between two points in the code, one way to do this is to use the `Environment.TickCount` property.[7] (This is more reliable than using the current date and time, because that can change as a result of the clock being adjusted, or when moving between time zones. The tick count just keeps increasing at a steady rate. That said, in real code you'd probably use the class library's `Stopwatch` class.) Example 2-34 shows one way to do this.

Example 2-34. Exploiting unchecked integer overflow

```
int start = Environment.TickCount;
DoSomeWork();
int end = Environment.TickCount;

int totalTicks = end - start;
Console.WriteLine(totalTicks);
```

The tricky thing about `Environment.TickCount` is that it occasionally 'wraps round'. It counts the number of milliseconds since the system last rebooted, and since its type is `int`, it will eventually run out of range. A span of 25 days is 2.16 billion milliseconds, too large a number to fit in an `int`. Imagine the tick count is 2,147,483,637, which is 10 short of the maximum value for `int`. What would you expect it to be 100ms later? It can't be 100 higher (2,147,483,727) because that's too big a value for an `int`. We'd expect it to get to the highest possible value after 10ms, so after 11ms it'll roll round to the minimum value, so after 100ms we'd expect the tick count to be 89 above the minimum value (which would be −2,147,483,559).

> The tick count is not necessarily precise to the nearest millisecond in practice. It often stands still for milliseconds at a time before leaping forward in increments of 10ms, 15ms, or even more. However, the value still rolls around, you just might not be able to observe every possible tick value as it does so.

Interestingly, Example 2-34 handles this perfectly. If the tick count in `start` was obtained just before the count wrapped, and the one in `end` was obtained just after, `end` will contain a much lower value than `start`, which seems backwards, and the difference between them will be large—larger than the range of an `int`. However, when we

[7] A property is a member of a type that represents a value that can either be read or modified or both; properties will be described in detail in Chapter 3.

subtract `start` from `end`, the overflow rolls over in a way that exactly matches the way the tick count rolls over, meaning we end up getting the correct result regardless. For example, if the `start` contains a tick count from 10ms before rollover, and `end` is from 90ms afterwards, subtracting the relevant tick counts (i.e. subtracting −2,147,483,558 from 2,147,483,627) seems like it should produce a result of 4,294,967,185, but because of the way the subtraction overflows, we actually get a result of 100, which corresponds to the elapsed time of 100ms.

But most of the time, this sort of integer overflow is undesirable. It means that when dealing with large numbers, you can get results that are completely incorrect. A lot of the time this is not a big risk because you'll be dealing with fairly small numbers, but if there's any possibility that your calculations might encounter overflow, you might want to use a checked context. Any arithmetic performed in a checked context will throw an exception when overflow occurs. You can request this in an expression with the `checked` operator, as Example 2-35 shows. Everything inside the parentheses will be evaluated in a checked context, so you'll see an `OverflowException` if the addition of `a` and `b` overflows. The `checked` keyword does not apply to the whole statement here, so if an overflow happens as a result of adding `c`, that will not cause an exception.

Example 2-35. Checked expression

```
int result = checked(a + b) + c;
```

You can also turn on checking for an entire block of code with a `checked` statement, which is a block preceded by the `checked` keyword, as Example 2-36 shows. Checked statements always involved a block—you could not just add the `checked` keyword in front of the `int` keyword in Example 2-35 to turn that into a checked statement. You'd also need to wrap the code in braces.

Example 2-36. Checked statement

```
checked
{
    int r1 = a + b;
    int r2 = r1 - (int) c;
}
```

C# also has an `unchecked` keyword. You can use this inside a checked block to indicate that a particular expression or nested block should not be a checked context. This makes life easier if you want everything except for one particular expression to be checked—rather than having to label everything except the chosen part as checked, you can put all the code into a checked block, and then exclude the one piece that wants to allow overflow without errors.

You can configure the C# compiler to put everything into a checked context by default, so that only explicitly `unchecked` expressions and statements will be able to overflow silently. In Visual Studio, you can configure this by opening the project properties, going to the Build tab, and clicking the `Advanced…` button. From the command line, you can use the compiler's `/checked` option. Be aware that there's a significant cost—checking can make individual integer operations several times slower. The impact on your application as a whole will be smaller, because programs don't spend their whole time performing arithmetic, but the cost may still be non-trivial. Of course, as with any performance matter, you should measure the practical impact. You may find that the performance cost is an acceptable price to pay for the guarantee that you will find out about unexpected overflows.

BigInteger

There's one last numeric type worth being aware of. `BigInteger` was introduced in .NET 4.0. It's part of the .NET Framework class library, and gets no special recognition from the C# compiler. However, it defines arithmetic operators and conversions, meaning that you can use it just like the built-in data types. It will compile to slightly less compact code—the compiled format for .NET programs can represent integers and floating point values natively, but `BigInteger` has to rely on the more general purpose mechanisms used by ordinary class library types. In theory it is likely to be significantly slower too, although in an awful lot of code, the speed at which you can perform basic arithmetic on integers is not a liming factor, so it's quite possible that you won't notice. And as far as the programming model goes, it looks and feels like a normal numeric type in your code.

As the name suggests, a `BigInteger` represents an integer. Its unique selling point is that it will grow as large as is necessary to accommodate values. So unlike the built-in numeric types, this has no theoretical limit on its range. Example 2-37 uses it to calculate values in the Fibonacci sequence, printing out every 100,000th value. This quickly produces numbers far too large to fit into any of the other integer types. I've shown the full source here to illustrate that this type is defined in the `System.Numerics` namespace. In fact `BigInteger` is in a separate DLL that Visual Studio does not reference by default, so you'll need to add a reference to the `System.Numerics` component to get this to run.

Example 2-37. Using BigInteger

```
using System;
using System.Numerics;

class Program
{
    static void Main(string[] args)
    {
        BigInteger i1 = 1;
        BigInteger i2 = 1;
        Console.WriteLine(i1);
        int count = 0;
        while (true)
        {
            if (count++ % 100000 == 0)
            {
                Console.WriteLine(i2);
            }
            BigInteger next = i1 + i2;
            i1 = i2;
            i2 = next;
        }
    }
}
```

Although `BigInteger` imposes no fixed limit, there are practical limits. You might produce a number that's too big to fit in the available memory, for example. Or more likely, the numbers may grow large enough that the amount of CPU time required to perform even basic arithmetic becomes prohibitive. But until you run out of either memory or patience, `BigInteger` will grow to accommodate numbers as large as you like.

Booleans

C# defines a type called `bool`, or as the runtime calls it, `System.Boolean`. This offers only two values: `true` and `false`. Whereas some C-family languages allow numeric types to stand in for Boolean values, with conventions such as 0 meaning true and anything else meaning false, C# will not accept a number. It demands that true/false values be represented by a `bool`. For example, in an `if` statement, you cannot write `if (someNumber)` to get some code to run only when `someNumber` is non-zero. If that's what you want, you need to say so explicitly by writing `if (someNumber != 0)`.

Strings and Characters

The `string` type (synonymous with the CLR `System.String` type) represents a sequence of text characters. Each character in the sequence is of type `char`, which is a 16-bit value representing a single UTF-16 code unit.

.NET strings are immutable. There are many operations that sound as though they will modify a string, such as concatenation, or the `ToUpper` and `ToLower` methods offered by instances of the `string` type, but all of these generate a new string, leaving the original one unmodified. This means that it's always safe to pass strings as arguments to code you didn't write, and you can be certain that your strings cannot be changed by that code.

The downside of immutability is that string processing can be inefficient. If you need to do work which performs a series of modifications to a string, such as building it up character by character, you will end up allocating a lot of memory, because you'll get a separate string for each modification. In these situations you can use a type called `StringBuilder`. (This is not a type that gets any special recognition from the C# compiler, unlike `string`.) This is conceptually similar to a `string`—it is a sequence of characters, and offers various useful string manipulation methods—but it is modifiable.

Object

The last intrinsic data type recognized by the C# compiler is `object` (or `System.Object` as the CLR calls it). This is the base class of almost[8] all C# types. A variable of type `object` is able to refer to a value of any type that derives from `object`. This includes all numeric types, the `bool` and `string` types, and any custom types you can define using the keywords we'll look at in the next chapter such as `class` and `struct`. And it also includes all the types defined by the .NET Framework class library.

So `object` is the ultimate general purpose container. You can refer to almost anything with an `object` variable. We shall return to this in Chapter 6 when we look at inheritance.

[8] There are some specialized exceptions, such as pointer types.

Operators

Earlier you saw that expressions are sequences of operators and operands. I've shown some of the types that can be used as operands, so now it's time to see what operators C# offers. Table 2-3 shows the operators that support common arithmetic operations.

Table 2-3. Basic arithmetic operators

Name	Example
Identity (unary plus)	`+x`
Negation (unary minus)	`-x`
Post-increment	`x++`
Post-decrement	`x--`
Pre-increment	`++x`
Pre-decrement	`--x`
Multiplication	`x * y`
Division	`x / y`
Remainder	`x % y`
Addition	`x + y`
Subtraction	`x - y`

If you're familiar with any other C family language, all of these should seem familiar. If you are not, the most peculiar ones will probably be the increment and decrement operators. These all have side effects: they add or subtract one from the variable to which they are applied (meaning they can only be applied to variables). With the post-increment and post-decrement, although the variable gets modified, the containing expression ends up getting the original value. So if `x` is a variable containing the value 5, the value of `x++` is also 5, even though the `x` variable will have a value of 6 after evaluating the `x++` expression. The pre- forms evaluate to the modified value, so if `x` is initially 5, `++x` evaluates to 6, which is also the value of `x` after evaluating the expression.

Although the operators in Table 2-3 are used in arithmetic, some are available on certain non-numeric types. As you saw earlier, the `+` symbol represents concatenation when working with strings. And as you'll see in Chapter 9, the addition and subtraction operators are also used for combining and removing delegates.

C# also offers some operators that perform certain binary operations on the bits that make up a value, shown in Table 2-4. C# does not support these operations on floating point types.

Table 2-4. Binary integer operators

Name	Example
Bitwise negation	`~x`
Bitwise AND	`x & y`
Bitwise OR	`x | y`

Name	Example
Bitwise XOR	`x ^ y`
Shift left	`x << y`
Shift right	`x >> y`

The bitwise negation operator inverts all bits in an integer—any binary digit with a value of 1 becomes 0 and vice versa. The shift operators move all the binary digits left or right by one position. A left shift sets the bottom digit to 0. Right shifts of unsigned integers set the top digit with zero, and right shifts of signed integers leave the top digit as it is, i.e., negative numbers remain negative because they keep their top bit set, while positive numbers keep their top bit as zero, so they also retain their sign.

The bitwise AND, OR, and XOR (exclusive OR) operators perform Boolean logic operations on each bit of the two operands, when applied to integers. These three operators are also available when the operands are of type `bool`. (It's as though these operators treat a `bool` as a 1-digit binary number.) There are some additional operators available for `bool` values, shown in Table 2-5. The `!` operator does to a `bool` what the `~` operator does to each of the bits in an integer.

Table 2-5. Operators for bool

Name	Example
Logical negation (also known as NOT)	`!x`
Conditional AND	`x && y`
Conditional OR	`x || y`

If you have not used other C-family languages, the conditional versions of the AND and OR operators may not be familiar to you. These only evaluate their second operands if necessary. For example, when evaluating `(a && b)`, if the expression `a` is `false`, the code generated by the compiler will not even attempt to evaluate `b`, because the result will be `false` no matter what value `b` has. Conversely, the conditional OR operator does not bother to evaluate its second operand if the first is `true`, because the result will be `true` regardless of the second operand's value. This is significant if the second operand's expression either has side effects (e.g., it includes a method invocation) or might produce an error. For example, you often see code of the form shown in Example 2-38.

Example 2-38. The conditional AND operator

```
if (s != null && s.Length > 10)
...
```

This checks to see if the variable `s` contains the special value `null`, meaning that it doesn't currently refer to any value. The use of the `&&` operator here is important, because if `s` is `null`, evaluating the expression `s.Length` would cause a runtime error. If we had used the `&` operator, the compiler would have generated code that always evaluates both operands, meaning that we would see a `NullReferenceException` at runtime if `s` is `null`, but by using the conditional operator, we avoid that, because the second operand, `s.Length > 10`, will only be evaluated if `s` is not `null`.

Example 2-38 tests to see if a property is greater than 10 by using the `>` operator. This is one of several *relational operators*, which allow us to make comparisons between values. They all take two operands, and produce a `bool` result. Table 2-6 shows these, and they are supported for all numeric types. Some operators are available on some other types too. For example, you can compare string values with the `==` and `!=` operators. (There is no built-in meaning for the other relational operators with `string` because different countries have different ideas about the order in which to sort strings. If you want to compare strings, .NET offers the `StringComparer` class, which requires you to select the rules by which you'd like your strings ordered.)

Table 2-6. Relational operators

Name	Example
Less than	`x < y`
Greater than	`x > y`
Less than or equal	`x <= y`
Greater than or equal	`x >= y`
Equal	`x == y`
Not equal	`x != y`

As with most C-family languages, the equality operator is a pair of equals symbols. This is because a single equals symbol also produces a valid expression, and it means something else: it's an assignment, and assignments are expressions too. This can lead to an unfortunate problem in C-family languages: it's all too easy to write `if (x = y)` when you meant `if (x == y)`. Fortunately, this will usually produce a compiler error in C#, because C# has a special type to represent Boolean values. In languages that allow numbers to stand in for Booleans, both of pieces of code are legal even if `x` and `y` are numbers. (The first means to assign the value of `y` into `x`, and then to execute the body of the `if` statement if that value is non-zero. That's very different than the second one, which doesn't change the value of anything, and only executes the body of the `if` statement if `x` and `y` are equal.) But in C#, the first example would only be meaningful if `x` and `y` were both of type `bool`.[9]

Another feature that's common to the C family is the conditional operator. (This is sometimes also called the ternary operator, because it's the only operator in the language that takes three operands.) It chooses between two expressions. More precisely, it evaluates its first operand, which must be a Boolean expression and then returns the value of either the second or third operand depending on whether the value of the first was true or false respectively. Example 2-39 uses this to pick the larger of two values. (This is just for illustration. In practice, you'd normally use .NET's `Math.Max` method which does the same thing, but is rather more readable.)

Example 2-39. The conditional operator

[9] Language pedants will note that this is not exactly right. It will also be meaningful in certain situations where custom implicit conversions to `bool` are available. We'll be getting to custom conversions in Chapter 3.

```
int max = (x > y) ? x : y;
```

This illustrates why C and its successors have a reputation for terse syntax. If you are familiar with any language from this family, Example 2-39 will be easy to read, but if you're not, its meaning might not be instantly clear. This will evaluate the expression before the `?` symbol, `(x > y)` in this case, and that's required to be an expression that produces a `bool`. If that is `true`, the expression between the `?` and `:` symbols is used (`x` in this case), and otherwise, the expression after the `:` symbol (`y` here) is used.

> The parentheses in Example 2-39 are optional. I put them in because I think they make the code easier to read.

The conditional operator is similar to the conditional AND and OR operators, in that it will only evaluate the operands it has to. It always evaluates its first operand, but it will never evaluate both the second and third operands. That means you can handle null values by writing something like Example 2-40. This does not risk causing a `NullReferenceException` because it will only evaluate the third operand if `s` is not `null`.

Example 2-40. Exploiting conditional evaluation

```
int characterCount = s == null ? 0 : s.Length;
```

However, in some cases, there are simpler ways of dealing with `null` values. Suppose you have a `string`, and if it's `null`, you'd like to use the empty string instead. You could write `(s == null ? "" : s)`. But you could just use the null coalescing operator instead, because it's designed for precisely this job. This operator, shown in Example 2-41 (it's the `??` symbol) evaluates its first operand, and if that's non-`null`, that's the result of the expression. But if the first operand is `null`, it evaluates its second operand, and uses that instead.

Example 2-41. The null coalescing operator

```
string neverNull = s ?? "";
```

One of the main benefits offered by both the conditional and the null coalescing operators is that they allow you to write a single expression in cases where you would otherwise have needed to write considerably more code. This can be particularly useful if you're using the expression as an argument to a method, as in Example 2-42.

Example 2-42. Conditional expression as method argument

```
FadeVolume(gateOpen ? MaxVolume : 0.0, FadeDuration, FadeCurve.Linear);
```

Compare this with what you'd need to write if the conditional operator did not exist. You would need an `if` statement. (I'll get to `if` statements in the next section, but since this book is not for novices, I'm assuming you're familiar with the rough idea.) And you'd either need to introduce a local variable as Example 2-43 does, or you'd need to duplicate the method call in the two branches of the `if`/`else`, changing just the first argument. So terse though the conditional and null coalescing operators are, once you're used to them they can remove a lot of clutter from your code.

Example 2-43. Life without the condition operator

```
double targetVolume;
if (gateOpen)
{
    targetVolume = MaxVolume;
```

```
}
else
{
    targetVolume = 0.0;
}
FadeVolume(targetVolume, FadeDuration, FadeCurve.Linear);
```

One last set of operators to look at are the *compound assignment* operators. These combine assignment with some other operation, and they are available for the `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `^`, and `|` operators. So you don't have to write the sort of code shown in Example 2-44.

Example 2-44. Assignment and addition

```
x = x + 1;
```

We can write this assignment statement more compactly as the code in Example 2-45. All the compound assignment operators take this form—you just stick an `=` on the end of the original operator.

Example 2-45. Compound assignment (addition)

```
x += 1;
```

As well as being more succinct, this can be less offensive to those of a sensitive mathematical disposition. Example 2-44 looks like a mathematical equation, but one that is complete nonsense. (This doesn't stop it being perfectly legal as C# of course—we are requesting an operation with side effects, rather than stating a truth. It only looks weird because C family languages use the `=` symbol to denote assignment rather than equality.) Example 2-45 doesn't even resemble any common basic mathematical notation. More usefully, it is a distinctive syntax that makes it very clear that we are modifying the value of a variable in some particular way. So although those two snippets perform identical work, many developers find the second idiomatically preferable.

That's not quite a comprehensive list of operators. There are a few more specialized ones that I'll get to once we've looked at the areas of the language for which they were defined. (Some relate to classes and other types, some to inheritance, some to collections, and some to delegates. There are chapters coming up on all of these.) By the way, although I've been describing which operators are available on which types (e.g., numeric vs. Boolean), it's possible to write a custom type that defines its own meanings for most of these. That's how the .NET Framework's `BigInteger` type can support the same arithmetic operations as the built-in numeric types. I'll show how to do this in Chapter 3.

Flow Control

Most the code we have examined so far executes statements in the order they are written, and stops when it reaches the end. If that were the only possible way in which execution could flow through our code, C# would not be very useful. So as you'd expect, it has a variety of constructs for writing loops, and for deciding which code to execute based on input conditions.

Boolean Decisions with if Statements

An `if` statement decides whether or not to run some particular statement depending on the value of a `bool` expression. For example, the `if` statement in Example 2-46 will only execute the block statement that prints a message if the `age` variable's value is less than 18.

Example 2-46. Simple if statement

```
if (age < 18)
{
    Console.WriteLine("You are too young to buy alcohol in a bar in the UK.");
}
```

You don't have to use a block statement with an `if` statement. You can use any statement type as the body. A block is only necessary if you want the `if` statement to govern the execution of multiple statements. However, many coding style guidelines recommend using a block in all cases. This is partly for consistency, but also because it avoids a possible error when modifying the code at a later date: if you have a non-block statement as the body of an `if`, and then you add an additional statement after that, intending it to be part of the same body, it can be easy to forget to wrap it in a block, leading to code like that in Example 2-47. The indentation suggests that the developer meant for the final statement to be part of the `if` statement's body, but C# ignores indentation, and so that final statement will always run. If you are in the habit of always using a block, you won't make this mistake.

Example 2-47. Probably not what was intended

```
if (launchCodesCorrect)
    TurnOnMissileLaunchedIndicator();
    LaunchMissiles();
```

An `if` statement can optionally include an `else` part, which is followed by another statement, which only runs if the `if` statement's expression evaluates to `false`. So Example 2-48 will print either the first or the second message, depending on whether the `optimistic` variable is true or false.

Example 2-48. If and else

```
if (optimistic)
{
    Console.WriteLine("Glass half full");
}
else
{
    Console.WriteLine("Glass half empty");
}
```

The `else` keyword can be followed by any statement, and again, this is typically a block. However, there's one scenario in which most developers do not use a block for the body of the `else` part, and that's when they use another `if` statement. Example 2-49 shows this—its first `if` statement has an `else` part, which has another `if` statement as its body.

Example 2-49. Picking one of several possibilities

```
if (temperatureInCelsius < 52)
{
```

```
    Console.WriteLine("Too cold");
}
else if (temperatureInCelsius > 58)
{
    Console.WriteLine("Too hot");
}
else
{
    Console.WriteLine("Just right");
}
```

This code still looks like it uses a block for that first `else`, but that block is actually the statement that forms the body of a second `if` statement. It's that second `if` statement that is the body of the `else`. If we were to stick rigidly to the rule of giving each `if` and `else` body its own block, we'd rewrite Example 2-49 as Example 2-50. This seems unnecessarily fussy, because the main risk that we're trying to avert by using blocks doesn't really apply in Example 2-49.

Example 2-50. Overdoing the blocks

```
if (temperatureInCelsius < 52)
{
    Console.WriteLine("Too cold");
}
else
{
    if (temperatureInCelsius > 58)
    {
        Console.WriteLine("Too hot");
    }
    else
    {
        Console.WriteLine("Just right");
    }
}
```

Although we can chain `if` statements together as shown in Example 2-49, C# offers a more specialized statement that can sometimes be easier to read.

Multiple Choice with switch Statements

A `switch` statement defines multiple groups of statements, and either runs one group, or does nothing at all, depending on the value of an expression. The expression can be any integer type, a `string`, a `char`, or any enumeration type (which we'll be looking at in Chapter 3). As Example 2-51 shows, you put the expression inside parentheses after the `switch` keyword, and after that there's a region delimited by braces containing series of `case` sections, defining the behavior for each anticipated value for the expression.

Example 2-51. A switch statement with strings

```
switch (workStatus)
{
case "ManagerInRoom":
    WorkDiligently();
    break;

case "HaveNonUrgentDeadline":
```

```
case "HaveImminentDeadline":
    CheckTwitter();
    CheckEmail();
    CheckTwitter();
    ContemplateGettingOnWithSomeWork();
    CheckTwitter();
    CheckTwitter();
    break;

case "DeadlineOvershot":
    WorkFuriously();
    break;


default:
    CheckTwitter();
    CheckEmail();
    break;
}
```

As you can see, a single section can serve multiple possibilities—you can put several different `case` lines at the start of a section, and the statements in that section will run if any of those cases apply. You can also write a `default` section, which will run if none of the cases apply. By the way, you're not required to provide a `default` section. A `switch` statement does not have to be comprehensive, so if there is no `case` that matches the expression's value, and there is no `default` section, the `switch` statement simply does nothing.

Unlike `if` statements, which take exactly one statement for the body, a `case` may be followed by multiple statements without needing to wrap them in a block. The sections in Example 2-51 are delimited by `break` statements, which causes execution to jump to the end of the `switch` statement. This is not the only way to finish a section—strictly speaking, the rule imposed by the C# compiler is that the end point of the statement list for each `case` must not be reachable, so anything that causes execution to leave the `switch` statement is acceptable. You could use a `return` statement instead, or throw an exception, or you could even use a `goto` statement.

Some C family languages (C, for example) allow *fall-through*, meaning that if execution is allowed to reach the end of the statements in a `case` section, execution will continue with the next one. Example 2-52 shows this style, and it is not allowed in C#, because of the rule that requires the end of a `case` statement list not to be reachable.

Example 2-52. C-style fall-through, illegal in C#

```
switch (x)
{
case "One":
    Console.WriteLine("One");
case "Two":  // This line will not compile
    Console.WriteLine("One or two");
    break;
}
```

C# outlaws this because the vast majority of `case` sections do not fall through, and when they do it's often a mistake caused by simply forgetting to write a `break` statement (or some other statement to break out of the `switch`). Accidental fall-through is likely to

produce unwanted behavior, so C# requires more than the mere omission of a `break`: if you want fall-through, you must ask for it explicitly. As Example 2-53 shows, we use the unloved `goto` keyword to express that we really do want one case to fall through into the next one.

Example 2-53. Fall through in C#

```
switch (x)
{
case "One":
    Console.WriteLine("One");
    goto case "Two";
case "Two":
    Console.WriteLine("One or two");
    break;
}
```

Incidentally, the `goto` statement is not strictly limited to `switch` statements—you can add labels to your code and jump around within your methods. However, `goto` is heavily frowned upon, so fall through seems to be the only use for it that is considered respectable in modern society.

Loops: while and do

C# supports the usual C family loop mechanisms. Example 2-54 shows a `while` loop. This takes a `bool` expression. It evaluates that expression, and if the result is true, it will execute the statement that follows. So far, this is just like an `if` statement, but the difference is that once the loop's embedded statement is complete, it then evaluates the expression again, and if it's true again, it will execute the embedded statement a second time. And it will keep doing this until the expression evaluates to `false`. As with `if` statements, the body of the loop does not need to be a block, but it usually is.

Example 2-54. A while loop

```
while (!reader.EndOfStream)
{
    Console.WriteLine(reader.ReadLine());
}
```

The body of the loop may decide to finish the loop early. A `break` statement will break out of the loop. It does not matter whether the `while` expression is `true` or `false`—executing a `break` statement will always terminate the loop.

C# also offers the `continue` statement. Like a `break` statement, this terminates the current iteration, but unlike `break`, it will then re-evaluate the `while` expression, so iteration may continue. Both `continue` and `break` jump straight to the end of the loop, but you could think of `continue` as jumping directly to the point just before the loop's closing `}`, while `break` jumps to the point just after. By the way, `continue` and `break` are also available for all of the other loop styles I'm about to show.

Because a `while` statement evaluates its expression before each iteration, it's possible for a `while` loop not to run its body at all. Sometimes, you may want to write a loop that runs at least once, only evaluating the `bool` expression after the first iteration. You can do this with a `do` loop, as shown in Example 2-55.

Example 2-55. A do loop

```
char k;
do
{
    Console.WriteLine("Press x to exit");
    k = Console.ReadKey().KeyChar;
}
while (k != 'x');
```

Notice that Example 2-55 ends in a semicolon, denoting the end of the statement. Compare this with the line containing the `while` keyword Example 2-54, which does not, despite otherwise looking very similar. That may look inconsistent, but it's not a typo. Putting a semicolon at the end of the line with the `while` keyword in Example 2-54 would be legal, but would change the meaning—it would indicate that we want the body of the `while` loop to be an empty statement. The block that followed would then be treated as a brand new statement to execute after the loop completes. The code would get stuck in an infinite loop unless the reader were already at the end of the stream. (The compiler will issue a warning about a "Possible mistaken empty statement" if you do that by the way.)

C-Style for Loops

Another style of loop that C# inherits from C is the `for` loop. This is similar to `while`, but it adds two features to that loop's `bool` expression: it provides a place to declare and/or initialize one or more variables that will remain in scope for as long as the loop runs, and it provides a place to perform some operation each time round the loop (in addition to the embedded statement that forms the body of the loop). So the structure of a `for` loop looks like this:

```
for (initializer; condition; iterator) body
```

A very common application of this is to do something to all the elements in an array. Example 2-56 shows a `for` loop that multiplies every element in an array by 2. The condition part works in exactly the same way as in a `while` loop—it determines whether the embedded statement forming the loop's body runs, and it will be evaluated before each iteration. Again, the body doesn't strictly have to be a block, but usually is.

Example 2-56. Modifying array elements with a for loop

```
for (int i = 0; i < myArray.Length; i++)
{
    myArray[i] *= 2;
}
```

The initializer in this example declares a variable called `i` and initializes it to 0. This initialization happens just once of course—this wouldn't be very useful if it reset the variable to 0 every time round the loop, because the loop would never end. This variable's lifetime effectively begins just before the loop starts, and finishes when the loop finishes. The initializer does not need to be a variable declaration—you can use any expression statement.

The iterator in Example 2-56 just adds 1 to the loop counter. It runs at the end of each loop iteration, after the body runs, and before the condition is re-evaluated. (So if the condition is initially false, not only does the body not run, the iterator will never be evaluated.) C# does nothing with the result of the iterator expression—it is useful only

for its side effects. So it doesn't matter whether you write `i++`, `++i`, `i += 1`, or even `i = i + 1`.

The iterator is a redundant construct, because it doesn't let you do anything that you couldn't have achieved by putting the exact same code in a statement at the end of the loop's body instead.[10] However, there may be readability benefits. A `for` statement puts the code that defines how we loop in one place, separate from the code that defines what we do each time round the loop, which might help someone reading the code to understand what it does. They don't have to scan down to the end of a long loop to find the iterator statement (although a long loop body that trails over pages of code is generally considered to be bad practice, so this is last benefit is a little dubious).

Both the initializer and the iterator can contain lists, as Example 2-57 shows, although this isn't terribly useful—it will run all iterators every time round, so in this example, `i` and `j` will have the same value as each other throughout.

Example 2-57. Multiple initializers and iterators

```
for (int i = 0, j = 0; i < myArray.Length; i++, j++)
...
```

You can't write a single `for` loop that performs a multi-dimensional iteration. If you want that, you would simply nest one loop inside another, as Example 2-58 illustrates.

Example 2-58. Nested for loops

```
for (int j = 0; j < height; ++j)
{
    for (int i = 0; i <  width; ++i)
    {
        ...
    }
}
```

Although Example 2-56 shows a common enough idiom for iterating through arrays, you will often use a different, more specialized construct.

Collection Iteration with foreach Loops

C# offers a style of loop which is not universal in C family languages. The `foreach` loop is designed for iterating through collections. A `foreach` loop fits this pattern:

```
foreach (item-type iteration-variable in collection) body
```

The *`collection`* is an expression whose type matches a particular pattern recognized by the compiler. The .NET Framework's `IEnumerable<T>` interface, which we'll be looking at in Chapter 5, matches this pattern, although the compiler doesn't actually require an implementation of that interface—it just requires the collection to implement a `GetEnumerator` method that resembles the one defined by that interface. Example 2-59 uses `foreach` to print all the strings in an array; all arrays provide the method that `foreach` requires.

[10] A `continue` statement complicates matters, because it provides a way to move to the next iteration without getting all the way to the end of the loop body. Even so, you could still reproduce the effect of the iterator when using `continue` statements, it would just require more work.

Example 2-59. Iterating over a collection with foreach

```
string[] messages = GetMessagesFromSomewhere();
foreach (string message in messages)
{
    Console.WriteLine(message);
}
```

This loop will run the body once for each item in the array. The *iteration variable* (`message`, in this example) is different each time round the loop, and will refer to the item for the current iteration.

In one way, this is less flexible than the `for`-based loop shown in Example 2-56: a `foreach` loop cannot modify the collection it iterates over. That's because not all collections support modification. `IEnumerable<T>` demands very little of its collections, as it does not require modifiability, random access, or even the ability to know up front how many items the collection provides. (In fact, `IEnumerable<T>` is able to support never-ending collections. For example, it's perfectly legal to write an implementation that returns random numbers for as long as you care to keep fetching values.)

But `foreach` offers two advantages over `for`. One advantage is subjective, and therefore debatable: it's slightly more readable. But significantly, it's also more general. If you're writing methods that do things to collections, those methods will be more broadly applicable if they use `foreach` than `for`, because you'll be able to accept an `IEnumerable<T>`. Example 2-60 can work with any collection that contains strings, rather than being limited to arrays.

Example 2-60. General purpose collection iteration

```
public static void ShowMessages(IEnumerable<string> messages)
{
    foreach (string message in messages)
    {
        Console.WriteLine(message);
    }
}
```

Summary

In this chapter, I showed the nuts and bolts of C# code—variables, statements, expressions, basic data types, operators, and flow control. Now it's time to take a look at the broader structure of a program. All code in C# programs must belong to a type, and types are the topic of the following chapter.

3

Types

C# does not limit us to the built-in data types shown in Chapter 2. You can define your own types. In fact, you have no choice: if you want to write code at all, C# requires you to define a type to contain that code. Everything we write, and any functionality we consume from the .NET Framework Class Library (or any other .NET library) will belong to a type.

C# recognizes multiple kinds of type. I'll begin with the most important.

Classes

Whether you're writing your own types, or using other people's, most of the types you work with in C# will be *classes*. A class can contain both code and data, and it can choose to make some of its features publicly accessible to anyone who cares to use it, while keeping other features only accessible to code within the class. So classes offer a mechanism for *encapsulation*—they can define a clear public programming interface for other people to use, while keeping internal implementation details inaccessible.

If you're familiar with object-oriented languages, this will all seem very ordinary. If you're not, then you might want to read a more introductory-level book first, because this book is not meant to teach programming. ("Learning C#" by Jesse Liberty, published by O'Reilly provides an introduction to the basic coding concepts used in C#.) So I'll just describe the details specific to C# classes.

You've already seen some examples of classes, but let's look at the structure in a bit more detail. Example 3-1 shows a simple class.

Example 3-1. A simple class

```
public class Counter
{
    private int _count;

    public int GetNextValue()
    {
```

```
        _count += 1;
        return _count;
    }
}
```

Class definitions always contain the `class` keyword followed by the name of the class. C# does not demand that the name matches the containing file, nor does it limit you to having one class in a file. Having said that, most C# projects make the class and file names match by convention. Class names follow the same rules for identifiers as variables, which I described in Chapter 2.

[I'd like to add a page reference—the relevant section is the paragraph immediately after Example 2-3]

The first line of Example 3-1 contains an additional keyword: `public`. Class definitions can optionally specify *accessibility*, which determines what other code is allowed to use the class. Ordinary classes have just two choices here: `public` and `internal`, with the latter being the default. (As I'll show later, you can nest classes inside other types, and nested classes have a slightly wider range of accessibility options.) An internal class is only available for use within the component that defines it. So if you are writing a class library, you are free to define classes that exist purely as part of your library's implementation: by marking them as `internal`, you prevent the rest of the world from using them.

> You can choose to make your internal types visible to selected external components. Microsoft does this with their libraries. The .NET Framework Class Library is spread across many DLLs, each of which defines many internal types, but some internal features are used by other DLLs from the library. This is made possible by annotating a component with the `[assembly: InternalsVisibleTo("<name>")]` attribute, specifying the name of the component with which you wish to share. (This would normally go in the *AssemblyInfo.cs* source file, which hides inside the *Properties* node in Solution Explorer.) For example, you might want to make every class in your application visible to a unit test project, so that you can write unit tests for code that you don't intend to make publicly available.

The `Counter` class in Example 3-1 has chosen to be `public`, but that doesn't mean it has to have everything on show. It defines two members—a field called `_count` that holds an `int`, and a method called `GetNextValue` that operates on the information in that field. As you can see, both of these members have accessibility qualifiers too. And as is very common with object-oriented programming, this class has chosen to make the data member private, exposing public functionality through a method.

Accessibility modifiers are optional for members, just as they are for classes, and again, they default to the most restrictive option available: `private` in this case. So I could have left off the `private` keyword in Example 3-1 without changing the meaning, but I prefer to be explicit. (If you leave it unspecified, people reading your code may wonder whether the omission was deliberate or accidental.)

Naming Conventions

Microsoft defines a set of conventions for publicly visible identifiers, which they (mostly) conform to in their class libraries. I usually follow these conventions in my code. You can get a free tool, FxCop which can verify, amongst other things, that a library conforms to these conventions. This ships as part of the Windows SDK, and is also built into the 'static analysis' tools that come with some editions of Visual Studio. Or if you just want to read a description of the rules, they're part of the design guidelines for .NET class libraries at http://msdn.microsoft.com/library/ms229042

In these conventions, the first letter of a class name is capitalized, and if the name contains multiple words, each new word is also capitalized. (For historical reasons, this convention is called Pascal Casing, or sometimes PascalCasing without a space, as a self-referential example.) Although it's legal in C# for identifiers to contain underscores, the conventions don't allow them in class names. Methods also use PascalCasing, as do properties. Fields are rarely public, but when they are, they use the same casing.

Methods parameters use a different convention known as camelCasing, in which upper case letters are used at the start of all but the first word. The name describes the way this convention produces one or more humps in the middle of the word.

Microsoft's naming conventions remain silent for implementation details. (The original purpose of these rules, and the FxCop tool was to ensure a consistent feel across the whole public API of the .NET Framework class library. The 'Fx' is short for Framework.) So there is no standard for how private fields are named. Example 3-1 uses an underscore prefix. I've done this because I like fields to look different from local variables, so that I can tell easily what sort of data my code is working with, and it can also help to avoid situations where method parameter names clash with field names. However, some people find this convention ugly and prefer not to distinguish fields visibly. Some other people find it insufficiently obvious and prefer the more in-your-face `m_` (a lowercase m followed by an underscore) prefix.

Fields hold data. They are a kind of variable, but unlike a local variable, whose scope and lifetime is determined by its containing method, a field is tied to its containing type. Example 3-1 has been able to refer to the `_count` field by its unqualified name, because fields are in scope within their defining class. But what about the lifetime? We know that each invocation of a method gets its own set of local variables. How many sets of a class's fields are there? There are a couple of choices, but in this case, it's one per instance. Example 3-2 uses the `Counter` class from Example 3-1 to illustrate this. Notice that this code is in a separate class, to demonstrate that we can use the `Counter` class's public method from other classes. By convention Visual Studio puts the program entry point, `Main`, in a class called `Program`, so I've done the same in this example.

[Production note: when laying out this example, I'd prefer to avoid a page break occurring in the lines that consist of Console.WriteLine if that's possible. I've put blank lines in to help show the different phases of operations, but they would just look a bit confusing and messy if split across a page break, as happened with one draft of this.]

Example 3-2. Using a custom class

```
class Program
{
    static void Main(string[] args)
```

```
    {
        Counter c1 = new Counter();
        Counter c2 = new Counter();
        Console.WriteLine("c1: " + c1.GetNextValue());
        Console.WriteLine("c1: " + c1.GetNextValue());
        Console.WriteLine("c1: " + c1.GetNextValue());

        Console.WriteLine("c2: " + c2.GetNextValue());

        Console.WriteLine("c1: " + c1.GetNextValue());
    }
}
```

This uses the `new` operator to create new instances of my class. Since I use `new` twice, I get two `Counter` objects, and each has its own `_count` field. So we get two independent counts, as the program's output shows:

```
c1: 1
c1: 2
c1: 3
c2: 1
c1: 4
```

As you'd expect, it begins counting up, and then a new sequence starts at 1 when we switch to the second counter. But when we go back to the first counter it carries on from where it left off. This demonstrates that each instance has its own `_count`. But what if we don't want that? Sometimes you will want to keep track of information that doesn't relate to any single object.

Static Members

We add the `static` keyword to a member declaration to declare that the member is not associated with any particular instance of the class. Example 3-3 shows a modified version of the `Counter` class from Example 3-1. I've added two new members, both static, for tracking and reporting counts across all instances.

Example 3-3. Class with static members

```
public class Counter
{
    private int _count;
    private static int _totalCount;

    public int GetNextValue()
    {
        _count += 1;
        _totalCount += 1;
        return _count;
    }

    public static int TotalCount
    {
        get
        {
            return _totalCount;
        }
    }
```

```
}
```

`TotalCount` reports the count, but it doesn't do any work—it just returns a value that the class keeps up to date, and as I'll explain later in the "Properties" section, this makes it an ideal candidate for being a property rather than a method. The static field `_totalCount` keeps track of the total number of calls to `GetNextValue`, unlike the non-static `_count`, which just tracks calls to the current instance. Notice that I'm free to use that static field inside `GetNextValue` in exactly the same way as I use the non-static `_count`. The difference in behavior is clear if I add the line of code shown in Example 3-4 to the end of the `Main` method in Example 3-2.

Example 3-4. Using a static property

```
Console.WriteLine(Counter.TotalCount);
```

This line prints out 5, the sum of the two counts. Notice that to access a static member, I just write *`ClassName.MemberName`*. In fact, Example 3-4 uses two static members—as well as my class's `TotalCount` property, it uses the `Console` class's static `WriteLine` method.

Because I've declared `TotalCount` as a static property, the code it contains only has access to other static members. If it tried to use the non-static `_count` field, or call the non-static `GetNextValue` method, the compiler would complain. Replacing `_totalCount` with `_count` in the `TotalCount` property results in this error:

```
error CS0120: An object reference is required for the non-static field, method,
 or property 'ConsoleApplication1.Counter._count'
```

Since non-static fields are associated with a particular instance of a class, C# needs to know which instance to use. With a non-static method or property, that'll be whichever instance the method or property itself was invoked on. So in Example 3-2, I wrote either `c1.GetNextValue()` or `c2.GetNextValue()` to choose which of my two objects to use. C# passed the reference stored in either `c1` or `c2` respectively as an implicit first argument. You can get hold of that using the `this` keyword, by the way. Example 3-5 shows an alternative way we could have written the first line of `GetNextValue` from Example 3-3, indicating explicitly that we believe `_count` is a member of the instance on which the `GetNextValue` method was invoked.

Example 3-5. The this keyword

```
this._count += 1;
```

Explicit member access through `this` is sometimes necessary due to name collisions. Although all the members of a class are in scope for any code in the same class, the code in a method does not share a declaration space with the class. Remember from Chapter 2 that a declaration space is a region of code in which a single name must not refer to two different entities, and since methods do not share theirs with the containing class, you are allowed to declare local variables and method parameters that have the same name as class members. This can easily happen if you don't use a convention such as an underscore prefix for field names. You don't get an error in this case—locals and parameters just hide the class members. But if you qualify access with `this`, you can get at class members even if there are locals with the same name in scope. Incidentally, some developers qualify all member access with `this`, presumably because they find the `_` and `m_` prefixes insufficiently obtrusive.

Of course, static methods don't get to use the `this` keyword, because they are not associated with any particular instance.

Static Classes

Some classes only provide static members. For example, the `System.Threading` namespace provides various classes that offer multithreading utilities. For example, there's the `Interlocked` class, which provides various atomic, lock-free, read-modify-write operations, and there's also the `LazyInitializer` class, which provides helper methods for performing deferred initialization in a way that guarantees to avoid double initialization in multithreaded environments. These classes only provide services through static methods. It makes no sense to create instances of these types because there's no useful per-instance information they could hold.

You can declare that your class is intended to be used this way by putting the `static` keyword in front of the `class` keyword. This compiles the class in a way that prevents instances of it from being constructed. Anyone attempting to construct instances of a class designed to be used this way clearly hasn't understood what it does, so the compiler error will be a useful prod in the direction of the documentation.

Reference Types

Any type defined with the `class` keyword will be a *reference type*, meaning that any variable of this type is merely a reference to an instance of the type. Consequently, assignments don't copy the object, they just copy the reference. Consider Example 3-6, which contains almost the same code as Example 3-2, except instead of using the `new` keyword to initialize the `c2` variable, it just initializes it with a copy of `c1`.

Example 3-6. Copying references

```
Counter c1 = new Counter();
Counter c2 = c1;
Console.WriteLine("c1: " + c1.GetNextValue());
Console.WriteLine("c1: " + c1.GetNextValue());
Console.WriteLine("c1: " + c1.GetNextValue());

Console.WriteLine("c2: " + c2.GetNextValue());

Console.WriteLine("c1: " + c1.GetNextValue());
```

Because this example uses `new` just once, there is only one `Counter` instance, and the two variables both refer to this same instance. So we get different output:

```
c1: 1
c1: 2
c1: 3
c2: 4
c1: 5
```

It's not just locals that do this—if you use a reference type as the type for any other kind of variable, such as a field or property, you will again see that assignment copies the reference, and not the whole object. This is different from the behavior we saw with the built-in numeric types in Chapter 2. With those, each variable contains a value, not a reference to a value, so assignment necessarily involves copying the value. This value

copying behavior is not available for most reference types—see the "Copying Instances" sidebar.

Copying Instances

Some C family languages define a standard way to make a copy of an object. For example, in C++ you can write a copy constructor, and you can overload the assignment operator, and there are rules for how these are applied when duplicating an object. In C#, some types are copyable, and it's not just the built-in numeric types. Later in this chapter you'll see how to define a *struct*, which is a custom value type. Structs are always copyable, but there is no way to customize this process: assignment just copies all the fields, and if any fields are of reference type, this just copies the reference. This is sometimes called a 'shallow' copy, because it only copies the contents of the struct, and does not make copies of any of the things the struct refers to.

There is no intrinsic mechanism for making a copy of a class instance. The .NET Framework does define an API for duplicating objects through its `ICloneable` interface, but this is not very widely supported. It's a problematic API because it doesn't specify how to handle objects with references to other objects. Should a clone also duplicate the objects to which it refers (a deep copy) or just copy the references (a shallow copy)? In practice, types that wish to allow themselves to be copied often just provide an ad hoc method for the job, rather than conforming to any pattern.

Now it would be possible to redesign `Counter` to make it feel a bit more like the built-in types. (Whether we *should* is questionable, but it'll be instructive to see where it takes us. We can assess whether it was a good idea when we get there.) One approach could make it *immutable*, meaning that it sets all of its fields during initialization and then never modifies them again. This is the tactic used by the built-in `string` type. You can ask the compiler to help you with this—if you use the `readonly` keyword in a field declaration, the compiler will generate an error if you attempt to modify that field from outside of a constructor.

Immutability doesn't give you copy-by-value semantics of course—assignment still just copies references, but if the object can never change state, then any particular reference will refer to a value that never changes, making it harder to tell the difference between copying a reference and copying a value. If you want to increment an immutable `Counter`, then you would need to produce a brand new instance, initialized with the incremented value. That's quite similar to how numbers work: an addition expression that adds `1` to an `int`, produces a brand new `int` value as the result.[1] You could achieve a similar effect by writing a custom implementation of the `++` operator for your own type. Example 3-7 shows how that might look.

Example 3-7. An immutable counter

```
public class Counter
{
    private readonly int _count;
```

[1] This does not necessarily require new memory, so this is more efficient than it sounds. New value instances often overwrite existing ones.

```
    private static int _totalCount;

    public Counter()
    {
        _count = 0;
    }

    private Counter(int count)
    {
        _count = count;
    }

    public Counter GetNextValue()
    {
        _totalCount += 1;
        return new Counter(_count + 1);
    }

    public static Counter operator ++(Counter input)
    {
        return input.GetNextValue();
    }

    public int Count
    {
        get
        {
            return _count;
        }
    }

    public static int TotalCount
    {
        get
        {
            return _totalCount;
        }
    }
}
```

I've had to modify the `GetNextValue` method to return a new instance, because it's no longer able to modify `_count`. This means my implementation of the `++` operator can just defer to `GetNextValue`. Example 3-8 shows how we can use this.

Example 3-8. Using an immutable counter

```
Counter c1 = new Counter();
Counter c2 = c1;
c1++;
Console.WriteLine("c1: " + c1.Count);
c1++;
Console.WriteLine("c1: " + c1.Count);
c1 = c1.GetNextValue();
Console.WriteLine("c1: " + c1.Count);

c2++;
Console.WriteLine("c2: " + c2.Count);
```

```
c1++;
Console.WriteLine("c1: " + c1.Count);
```

Notice that the code now only uses the `new` operator once. So after the `c2` is declared it holds a reference to the same object that `c1` refers to. But because these are immutable objects, we've had to change the way we update the counter. I can use either the `++` operator or `GetNextValue`, but in either case we end up creating a new instance of the type, and the reference that was previously in the variable is replaced with a reference to this new object. (Unlike with `int`, this new instance will always involve allocating new memory, but I'll show how to change that in the "Structs" section.) So although `c1` and `c2` started out referring to the same object as they did in Example 3-6, this time the output shows that we still get two independent sequences:

```
c1: 1
c1: 2
c1: 3
c2: 1
c1: 4
```

Of course, all that's happening here is that the `new` keyword is getting used multiple times, it's just hiding in the `++` operator and `GetNextValue` method. Conceptually, that's not very different from the fact that incrementing an integer produces a new integer value that is one higher; the number 5 does not stop being the number 5 just because you decided to calculate 5+1, just as a `Counter` with a count of 5 doesn't stop having that count just because you decided to ask for its successor.

However, there is one big difference between how immutable objects and the intrinsic numeric values work. Any single instance of a reference type has an identity, by which I mean that it is possible to ask whether two references refer to the exact same instance. I could have two variables that each refer to `Counter` objects which have a count of 1, which might mean they refer to the same `Counter`, but it's possible that they refer to different objects that happen to have the same value.

Example 3-9 arranges for three variables to refer to counters with the same count, and then compares their identities. By default, the `==` operator does exactly this sort of object identity comparison when its operands are reference types. However, types are allowed to redefine the `==` operator. The `string` type changes `==` to perform value comparisons, so if you pass two distinct string objects as the operands of `==`, the result will be true if they contain identical text. If you want to force comparison of object identity, you can use the static `object.ReferenceEquals` method.

Example 3-9. Comparing references

```
Counter c1 = new Counter();
c1++;
Counter c2 = c1;
Counter c3 = new Counter();
c3++;

Console.WriteLine(c1.Count);
Console.WriteLine(c2.Count);
Console.WriteLine(c3.Count);
Console.WriteLine(c1 == c2);
Console.WriteLine(c1 == c3);
Console.WriteLine(c2 == c3);
```

```
Console.WriteLine(object.ReferenceEquals(c1, c2));
Console.WriteLine(object.ReferenceEquals(c1, c3));
Console.WriteLine(object.ReferenceEquals(c2, c3));
```

The first three lines of output confirm that all three counters have the same count:

```
1
1
1
True
False
False
True
False
False
```

It also illustrates that while they have the same count, only `c1` and `c2` are considered to be the same thing. That because after incrementing `c1`, we assign it into `c2`, meaning that `c1` and `c2` will both refer to the same object, which is why the first comparison succeeds. But `c3` refers to a different object entirely that happens to have the same value, which is why the second comparison fails. (I've used both the `==` and `object.ReferenceEquals` comparisons here to illustrate that they do the same thing in this case, because `Counter` has not defined a custom meaning for `==`.)

We could try the same thing with `int` instead of a `Counter`, as Example 3-10 shows.

Example 3-10. Comparing values

```
int c1 = new int();
c1++;
int c2 = c1;
int c3 = new int();
c3++;

Console.WriteLine(c1);
Console.WriteLine(c2);
Console.WriteLine(c3);
Console.WriteLine(c1 == c2);
Console.WriteLine(c1 == c3);
Console.WriteLine(c2 == c3);
Console.WriteLine(object.ReferenceEquals(c1, c2));
Console.WriteLine(object.ReferenceEquals(c1, c3));
Console.WriteLine(object.ReferenceEquals(c2, c3));
Console.WriteLine(object.ReferenceEquals(c1, c1));
```

As before, we can see that all three variables have the same value:

```
1
1
1
True
True
True
False
False
False
False
```

This also illustrates that the `int` type does define a special meaning for `==`. This compares the values, so those three comparisons succeed. But `object.ReferenceEquals` never succeeds for value types—in fact, I've added an extra, fourth comparison here, where I compare `c1` with `c1`, and even that fails! That surprising result is due to the fact that it's not even meaningful to perform a reference comparison with `int`, because it's not a reference type. The compiler has had to perform implicit conversions for the last three lines of Example 3-10: it has wrapped each argument to `object.ReferenceEquals` in something called a box, and we'll be looking at those in Chapter 7.

There's another difference between reference types and types like `int` that's rather easier to demonstrate. Any reference type variable can contain a special value, `null`, meaning that the variable does not refer to any object at all. You cannot assign this value into any of the built-in numeric types (although see the sidebar, "Nullable<T>").

Nullable<T>

.NET defines a wrapper type called `Nullable<T>`, which adds nullability to value types. Although an `int` variable cannot hold `null`, a `Nullable<int>` can. The angle brackets after the type name indicate that this is a generic type—you can plug in various different types into that `T` placeholder—and I'll talk about those more in Chapter 4.

The compiler provides special handling for `Nullable<T>`. It lets you use a more compact syntax, so you can write `int?` instead. C# has special handling for nullable numerics inside arithmetic expressions. For example, if you write `a + b` where `a` and `b` are both `int?`, the result is an `int?` which will be `null` if either operand was `null`, and will otherwise contain the sum of the values. This also works if only one of the operands is an `int?` and the other is an ordinary `int`.

While you can set an `int?` to `null`, it's not a reference type. It's more like a combination of an `int` and a `bool`.

The difference between our immutable class and `int` clearly illustrates that the built-in numeric types are not the same sort of thing as a class. A variable of type `int` is not a reference to an `int`. It contains the value of the `int`—there is no indirection. In some languages, this choice between reference-like and value-like behavior is determined by the way in which you use a type, but in C#, this is a fixed feature of the type. Any particular type is either a reference type or a *value type*. The built-in numeric types are all value types, as is `bool` whereas a `class` is always a reference type. But this is not a distinction between built-in and custom types. You can write custom value types.

Structs

Sometimes it will be appropriate for a custom type to get the same value-like behavior as the built-in numeric types. The most obvious example would be a custom numeric type. For example, although the CLR has intrinsic support for some numeric types, we saw that the class library adds the `BigInteger` type for representing arbitrarily large integers. The same part of the library also defines a `Complex` type for representing complex numbers. It would be unhelpful if these types behaved significantly differently from the

built-in types. Fortunately, they don't, because they are value types. The way to write a custom value type is to use the `struct` keyword.

A struct can have most of the same features as a class—it can contain methods, fields, properties, constructors, and any of the other members types supported by classes, and we can use the same accessibility keywords such as `public` and `internal`. However, there are a few restrictions, so if we wanted to turn the `Counter` type I wrote earlier into a struct, we can't just replace the `class` keyword with `struct`. (Again, whether we *should* convert it to a struct is questionable. I'll return to that once we've done it.)

Slightly surprisingly, we'd need to remove the constructor which takes no arguments. The compiler always automatically provides a `struct` with a zero-argument constructor, and it is an error to attempt to provide your own. (This only applies to zero-argument constructors—you're allowed to define constructors that take arguments.) This compiler-generated constructor for a `struct` initializes all fields to 0, or the nearest equivalent value (e.g., `false` for a `bool` field, or `null` for a reference). This makes initialization of values very straightforward for the CLR. If you declare an array of some value type (whether a built-in type or a custom one), the array's values go in a single contiguous block of memory.[2] This is very efficient—for a large array, overhead such as heap block headers will take a tiny proportion of the space, with the bulk of the block containing the data you care about. Because value types are compelled to have a zero-argument constructor that does nothing more than set everything to 0, the entire array can be initialized quickly with a loop that fills it with zeros. The same is true for when a value type appears as a field in some other type—the memory for a newly allocated object gets filled with zeros, which has the effect of setting all reference-type fields to null, and all values to their default state. Not only is this efficient, it also simplifies initialization—constructors containing code will only run if you invoke them explicitly.

Looking at Example 3-7, our `Counter` class's no-arguments constructor initializes the one and only non-static field to 0, so the compiler-generated constructor we get with a `struct` does what we want anyway. So if we convert `Counter` to a `struct`, we can just remove that constructor and we won't lose anything.

We'll need to make one more change, or rather, a set of changes with one goal in mind. As mentioned earlier, C# defines a default meaning for the `==` operator for reference types: it is equivalent to `object.ReferenceEquals`, which compares identities. That's not meaningful for value types, so C# does not define any automatic meaning for `==` for a `struct`. We're not required to define a meaning, but if you write code that attempts to compare values with `==`, the compiler will complain if the type hasn't defined an `==` operator. However, if you add an `==` operator on its own, the compiler will inform you that you are required to define a matching `!=` operator. You might think C# would define `!=` as the inverse of `==`, since they appear to mean the opposite. However, some types will, in some situations return `false` for both operators for certain pairs of operands, so C# requires us to define both independently. As Example 3-11 shows, these are very straightforward for our simple type.

Example 3-11. Support custom comparison

[2] This is an implementation detail by the way, rather than an absolute requirement of how C# has to work, but it's how Microsoft's implementation currently works.

```
public static bool operator ==(Counter x, Counter y)
{
    return x.Count == y.Count;
}

public static bool operator !=(Counter x, Counter y)
{
    return x.Count != y.Count;
}

public override bool Equals(object obj)
{
    if (obj is Counter)
    {
        Counter c = (Counter) obj;
        return c.Count == this.Count;
    }
    else
    {
        return false;
    }
}

public override int GetHashCode()
{
    return _count;
}
```

If you just add the `==` and `!=` operators, you'll find that the compiler generates warnings recommending that you define two methods called `Equals` and `GetHashCode`. `Equals` is a standard method that is available on all .NET types, and if you have defined a custom meaning for `==`, you should ensure that `Equals` does the same thing. Example 3-11 does this, and as you can see, it contains the same logic as the `==` operator, but it has to do some extra work. The `Equals` method permits comparison with absolutely any type, so we first check to see if our `Counter` is being compared with another `Counter`. This involves some conversion operators that I'll be describing in more detail in Chapter 6.I'm using the `is` operator, which tests to see whether a variable refers to an instance of the specified type, and having established that our `Counter` is definitely being compared with another `Counter`, the `(Counter) obj` expression that follows lets us get hold of the `Counter` that `obj` refers to, enabling us to perform the comparison. Finally, Example 3-11 implements `GetHashCode`, which we're required to do if we implement `Equals`. See the "GetHashCode" sidebar for details.

GetHashCode

All .NET types offer a method called `GetHashCode`. It returns an `int` that in some sense represents the value of your object. Some data structures and algorithms are designed to work with this sort of simplified, reduced version of an object's value. A hash table, for example, can find a particular entry in a very large table very efficiently, as long as the type of value you're searching for offers a good hash code implementation. Some of the collection classes described in Chapter 5 rely on this. The details of this sort of algorithm are beyond the scope of this book, but if you search the web for "hash table" you'll find plenty of information.

A correct implementation of `GetHashCode` must meet two requirements. The first is that whatever number an instance returns as its hash code, that instance must continue to return the same code as long as its own value does not change. The second requirement is that two instances that have equal values according to their `Equals` methods *must* return the same hash code. Any type that fails to meet either of these requirements will cause code that uses its `GetHashCode` method to malfunction. The default implementation of `GetHashCode` meets the first requirement but makes no attempt to meet the second—pick any two objects that use the default implementation and most of the time they'll have different hash codes. This is why you need to override `GetHashCode` if you override `Equals`.

Ideally, objects that have different values should have different hash codes. Of course, that's not always possible—`GetHashCode` returns an `int`, which has a finite number of possible values. (4,294,967,296 to be precise.) If your data type offers more distinct values, then it's clearly not possible for every conceivable value to produce a different hash code. For example, the 64-bit integer type, `long`, obviously supports more distinct values than `int`. If you call `GetHashCode` on a `long` with a value of 0, on .NET 4.0 it returns 0, and you'll get the same hash code for a `long` with a value of 4,294,967,297. This is called a *hash collision*, and they are an unavoidable fact of life. Code that depends on hash codes just has to be able to deal with these.

The rules do not require the mapping from values to hash codes to be fixed forever. Just because a particular value produced a particular hash code today does not mean you can expect to get the same code for the same value when running your program next week. Nor are programs obliged to produce the same hash for the same value when running simultaneously on two different computers. In fact there are good reasons to avoid that. Criminals who attack online computer systems sometimes try to cause hash collisions. Collisions decrease the efficiency of hash-based algorithms, so an attack that attempts to overwhelm a server's CPU will be more effective if it can induce collisions for values that it knows the server will use in hash-based lookups. Some types in the .NET Framework deliberately change the way they produces hashes each time you restart a program to avoid this problem.

Because hash collisions are unavoidable, the rules cannot forbid them, which means you could return the same value from `GetHashCode` every time, regardless of the instance's actual value. So if you always return 0, for example, that's not technically against the rules. It will however tend to produce lousy performance from hash tables and the like. Ideally you will want to minimize hash collisions. That said, if you don't expect anything to depend on your type's hash code, there's not much point in spending time carefully devising a hash function that produces well-distributed values. Sometimes a lazy approach, such as deferring to a single field like Example 3-11 does, is OK.

With the modifications in Example 3-11 applied to the class in Example 3-7, and with the first constructor removed, we can change it from a `class` to a `struct`. Running the code in Example 3-9 one more time produces this output:

```
1
1
1
```

```
True
True
True
False
False
False
```

As before, all three counters have a count of 1, which shouldn't be any surprise. Then we have the first three comparisons which, remember, use `==`. Since Example 3-11 defines a custom implementation that compares values, it should be no surprise to see all the comparisons now succeed. And all of the `object.ReferenceEquals` values fail because this is now a value type, just like `int`. In fact this is the same behavior we saw with the code that used `int` instead of `Counter`. Variables of type `Counter` no longer hold a reference—they hold the value directly, so reference comparisons are no longer meaningful. (Again, the compiler has actually generated implicit conversions here that produce boxes, which we will look at in Chapter 7.)

It's time to ask an important question: was it a good idea to turn `Counter` into a value type? The answer is no. I've been hinting at that all along, but I wanted to illustrate some of the problems that can arise if you make something a struct inappropriately. So what does make a good struct?

When to Write a Value Type

I've shown some of the differences in observable behavior between a `class` and a `struct`, and I've illustrated some of the things you need to do differently to write a `struct`, but I've not yet explained how to decide which to use. The short answer is that there are only two circumstances in which you should write a value type. First, if you need to represent something value-like, such as a number, a struct is likely to be ideal. Second, if you have determined that a struct has usefully better performance characteristics for the scenario in which you will use the type, a struct may not be ideal, but might still be a good choice. But it's worth understanding the pros and cons in slightly more detail. And I will also dispel a surprisingly persistent myth about value types.

With reference types, the variable is a distinct entity from the object to which it refers. This can be very useful, because we often use objects as models for real things with identities of their own. But it has some performance implications. An object's lifetime is not necessarily directly related to the lifetime of a variable that refers to it. You can create a new object, and then pass a reference to that object to a method which might store that reference in a field of some other object which is itself referred to by some static field. The method that originally created the object might then return, so the local variable that first referred to the object no longer exists, but the object needs to stay alive because it's still possible to reach it by other means.

The CLR goes to considerable lengths to ensure that the memory an object occupies is not reclaimed prematurely, but that it is eventually freed once the object is no longer in use. This is a fairly complex process, and .NET applications can end up consuming a considerable amount of CPU time just tracking objects in order to work out when they fall out of use. Creating lots of objects increases this overhead. Increasing complexity in certain ways can also increase the costs of object tracking—if a particular object only remains alive because it is reachable through some very convoluted path, the CLR may need to follow that path each time it tries to work out what memory is still in use. Each

level of indirection you add generates extra work. A reference is by definition indirect, so every reference type variable creates additional work for the CLR.

Value types can often be handled in a much simpler way. For example, consider arrays. If you declare an array of some reference type, you end up with an array of references. This is very flexible—elements can be null if you want, and you're also free to have multiple different elements all referring to the same item. But if what you actually need is a simple sequential collection of items, that flexibility is just overhead. A collection of 1000 reference type instances requires 1001 blocks of memory: one block to hold an array of references, and then 1000 objects for those references to refer to. But with value types, a single block can hold all the values. This makes things simple for memory management purposes—either the array is still in use or it isn't, and there's no need to go on to check the 1000 individual elements separately.

It's not just arrays that can benefit from this sort of efficiency. Fields can also benefit. Consider a class that contains 10 fields all of type `int`. The 40 bytes required to hold those fields' values can live directly inside the memory allocated for an instance of the containing class. Compare that with 10 fields of some reference type. Although those references can be stored inside the object instance's memory, the objects they refer to will be separate objects, so if the fields are all non-null and all refer to different objects, you'll now have 11 blocks of memory—one for the instance that contains all the fields, and then one for each of the objects those fields refer to. Figure 3-1 illustrates these differences between references and values for both arrays and objects (with smaller examples, because the same principle applies even with a handful of instances).

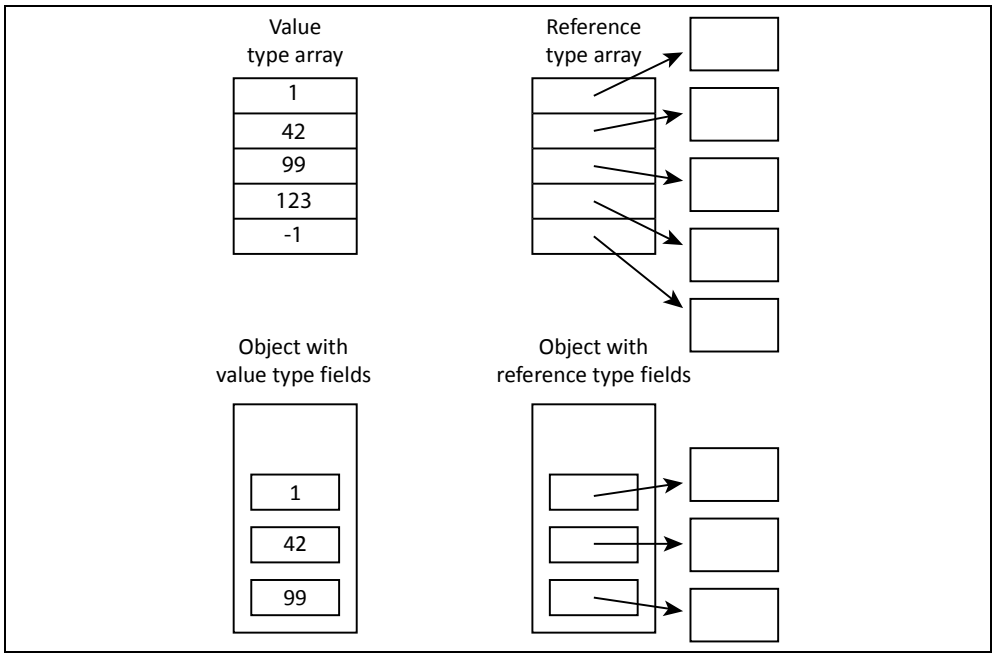

Figure 3-1. References vs values

Also, lifetime handling can sometimes be simpler for value types. Often, the memory allocated for local variables can be freed as soon as a method returns (although as we'll see in Chapter 9, nested methods mean that it's not always that simple). This means the memory for local variables can sometimes live on the stack, or even inside the CPU's

registers, which is typically much cheaper than the heap. For reference types, the memory for a variable is only part of the story—the object it refers to cannot be handled so easily, because that may continue to be reachable by other paths after the method exits. But with value types, the variable contains the value, so value types are better able to exploit the situations where memory for local variables can be handled efficiently.

In fact, the memory for a value may be reclaimed even before a method returns. New value instances often just overwrite older instances. For example, C# can normally just use a single piece of memory to represent a variable, no matter how many different values you put in there. Creating a new instance of a value type doesn't necessarily have to mean allocating more memory with value types, whereas with reference types, a new instance means a new heap block. This is why it's OK for each operation we perform with a value type—every integer addition or subtraction for example—to produce a new instance.

> One of the most persistent myths about value types says that values are allocated on the stack, unlike objects. It's true that objects always live on the heap, but value types don't always live on the stack. (And even in the situations where they do, that's an implementation detail, not a fundamental feature of C#.) Figure 3-1 shows two counterexamples. An `int` value inside an array of type `int[]` does not live on the stack; it lives inside the array's heap block. Likewise, if a class declares a non-static `int` field, the value of that `int` lives inside the heap block for its containing object instance. And even local variables of value type don't necessarily end up on the stack. For example, optimizations may make it possible for the value of a local variable to live entirely inside the CPU's registers, rather than needing to go on the stack.

You might be tempted to summarize the preceding few paragraphs as "there are some complex details, but in essence, value types are more efficient." But that would be a mistake. There are some situations in which value types are significantly more expensive. Remember that a defining feature of a value type is that values get copied on assignment. If the value type is big, that will be expensive. For example, the .NET Framework Class Library defines the `Guid` type to represent the 16-byte globally unique identifiers that crop up in lots of bits of Windows. This is a `struct`, so any assignment statement involving a `Guid` is asking to make a copy of a 16-byte data structure. This is likely to be more expensive than making a copy of a reference, because the CLR uses a pointer-based implementation for references, meaning that you'll be working something pointer-sized. (That's typically 4 or 8 bytes, but more importantly, it'll be something that fits naturally into a single CPU register.)

It's not just assignment that causes values to be copied. Passing a value type argument to a method may require a copy. As it happens, with method invocation it is actually possible to pass a reference to a value, although as we'll see later, it's a slightly limited kind of reference, and the restrictions it imposes are sometimes undesirable, so you may end up deciding that the cost of the copy is preferable.

Value types are not automatically going to be more efficient than reference types, so your choice should typically be driven by the behavior you require. The most important question is this: does the identity of an instance matter to you? In other words, is the distinction between one object and another object important? For our `Counter` example, the answer is probably yes: if we want something to keep count for us, it's simplest if that

counter is a distinct thing with its own identity. (Otherwise, our `Counter` type is no more useful than an `int`.) The code started getting strange as we moved away from that model. The original code in Example 3-3 was simpler than where we ended up.

An important and related question is: does an instance of your type contain state that changes over time? Most value types are immutable. This doesn't mean that variables of these types cannot be modified; it just means that to modify the variable, you must replace its contents entirely with a different value. For something simple like an `int` this will seem like splitting hairs, but the distinction is important with structs that contain multiple fields, such as the `Complex` type, which represents numbers that combine a real and an imaginary component. You cannot change the `Imaginary` property of an existing `Complex` instance, because the type is immutable. If the value you've got isn't the value you want, immutability just means you need to create a new value that is the one you want because you can't tweak the existing instance.

Immutability does not necessarily mean you should write a struct—the built-in `string` type is immutable, and that's a class.[3] However, because C# often does not need to allocate new memory to hold new instances of a value type, value types are able to support immutability more efficiently than classes in scenarios where you're creating lots of new values (e.g., in a loop). Immutability is not an absolute requirement for structs—there are some unfortunate exceptions in .NET's class library. But mutability tends to cause problems with value types, because it's all too easy to end up modifying some copy of the value rather than the instance you wanted to modify. Since value types should normally be immutable, a requirement for mutability is usually a good sign that you want a class rather than a struct. My `Counter` type started getting weird when I made it immutable—it was natural for it to maintain a count that changed over time, and it became much harder to use when we needed a whole new instance each time we wanted to change the count. This is another sign that `Counter` should be a class, not a struct.

A type should only be a struct if it represents something that is very clearly similar in nature to other things that are value types. For example, in the .NET Framework class library, `BigInteger` is a struct, which is unsurprising because it's a numeric type, and all of the built-in numeric types are value types. `TimeSpan`, is also a value type, which makes sense because it's effectively just a number that happens to represent a length of time. In the UI framework WPF, types used for simple geometric data such as `Point` and `Rect` are structs. But if in doubt, write a class; `Counter` was more usable as a class.

Members

Whether you're writing a class or a struct, there are several different kinds of members you can put in a custom type. We've seen examples of some already, but let's take a closer and more comprehensive look.

[3] The `string` type cannot be a struct because strings vary in length. However, that's not a factor you need to consider because you can't write your own variable-length data types in C#. Only with strings and array types can two instances of the same type have different sizes.

With one exception (static constructors) you can specify the accessibility for all class and struct members. Just as a type can be `public` or `internal`, so can each member. Members may also be declared as `private`, making them accessible only to code inside the type, and this is the default accessibility. And we'll see in Chapter 6, inheritance adds two more accessibility levels for members, `protected` and `protected internal`.

Fields

You've already seen that fields are named storage locations that hold either values or references depending on their type. By default, each instance of a type gets its own set of fields, but if you want a field to be singular, rather than having one per instance, you can use the `static` keyword. We've also seen the `readonly` keyword, which states that the field can only be set during construction, and cannot change thereafter.

> The `readonly` keyword does not make any absolute guarantees. There are mechanisms by which it is possible to contrive a change in the value of a `readonly` field. The reflection mechanisms in Chapter 13 provide one way to subvert `readonly`, and unsafe code, described in Chapter 23, provides another. The compiler will prevent you from modifying a field accidentally, but with sufficient determination, you can bypass this protection. And even without such subterfuge, a `readonly` field is free to change during construction.

C# offers a keyword which seems, superficially, to be similar: you can define a `const` field. However, this is designed for a somewhat different purpose. A `readonly` field is initialized and then never changed, whereas a `const` field defines a value that is invariably the same. A `readonly` field is much more flexible: it can be of any type, and its value can be calculated at runtime. A `const` field's value is determined at compile time, which limits the available values. For most reference types, the only supported `const` value is `null`, so in practice, it's normally only useful to use `const` with types intrinsically supported by the compiler. (Specifically, if you want to use values other than `null`, a `const` must be either one of the built-in numeric types, a `bool`, a `string`, or an enumeration type as described later in this chapter.)

This makes a `const` field rather more limited than a `readonly` one, so you could reasonably ask: what's the point? Well although a `const` field is inflexible, it makes a strong statement about the unchanging nature of the value. For example, the .NET Framework's `Math` class defines a `const` field of type `double` called `PI` that contains as close an approximation to the mathematical constant π as a `double` can represent. That's a value that's fixed forever—it is a constant in a very broad sense.

You need to be a bit careful about `const` fields—the compiler is allowed to assume that the value really will never change. Code that reads the value from a `readonly` field will fetch the value from the memory containing the field at runtime. But when you use a `const` field, the compiler is allowed to read the value at compile time and copy it into your code as though it were a literal. So if you write a library component that declares a `const` field and you later change its value, this change will not necessarily be picked up by code using your library unless that code gets recompiled.

One of the benefits of a `const` field is that it is eligible for use in certain contexts in which a `readonly` field is not. For example, the label for a `case` in a `switch`

statement has to be fixed at compile time, so it cannot refer to a `readonly` field, but you can define a `case` in terms of a suitably-typed `const` field. You can also use `const` fields in the expression defining the value of other `const` fields (as long as you don't introduce any circular references).

A `const` field is required to contain an expression defining its value, such as the one shown in Example 3-12.

Example 3-12. A const field

```
const double kilometersPerMile = 1.609344;
```

This initializer expression is optional for a class's ordinary and `readonly` fields. If you omit the initializing expression, the field will automatically be initialized to a default value. (That's zero for numeric values, and the equivalents for other types—`false`, `null`, etc.) Structs are slightly more limited, because their default initialization always involves setting all its instance fields to zero, so you are obliged to omit initializers for those. Structs do support initializers for non-instance fields though, i.e., `const` and `static` fields.

If you do supply an initializer expression for a non-`const` field, it does not need to be evaluable at compile time, so it can do runtime work like calling methods or reading properties. Of course, that sort of code can have side effects, so it's important to be aware of the order in which this sort of code runs.

Non-static field initializers run for each instance you create, and they execute in the order in which they appear in the file, immediately before the constructor runs. Static fields execute no more than once no matter how many instances of the type you create. They also execute in the order in which they are declared, but it's harder to pin down exactly when they will run. If your class has no static constructor, C# guarantees to run field initializers before the first time a field in the class is accessed, but it doesn't necessarily wait until the last minute—it retains the right to run field initializers as early as it likes. In fact, the exact moment at which this happens has varied across releases of Microsoft's C# implementation. But if a static constructor does exist, then things are slightly clearer: static field initializers run immediately before the static constructor runs, but that merely raises the questions: what's a static constructor, and when does that run? So we had better take a look at constructors.

Constructors

A newly created object may require some information to do its job. For example, The `Uri` class in the `System` namespace represents a Uniform Resource Identifier (URI) such as a URL. Since its entire purpose is to contain and provide information about a URI, there wouldn't be much point in having a `Uri` object that didn't know what its URI was. So it's not actually possible to create one without providing a URI. If you try the code in Example 3-13, you'll get a compiler error.

Example 3-13. Error: failing to provide a Uri with its URI

```
Uri oops = new Uri();  // Will fail to compile
```

The `Uri` class defines several *constructors*, members that contain code that initializes a new instance of a type. If a particular class requires certain information to work, you can

enforce this requirement through constructors. Creating an instance of a class almost always involves using a constructor at some point,[4] so if the constructors you define all demand certain information, developers will have to provide that information if they want to use your class. So all of the `Uri` class's constructors need to be given the URI in one form or another.

To define a constructor, you first specify the accessibility (`public`, `private`, `internal`, etc.) and then the name of the containing type. This is followed by a list of parameters in parentheses (which is allowed to be empty). Example 3-14 shows a class that defines a single constructor that requires two arguments: one of type `decimal`, and one of type `string`. The argument list is followed by a block containing code. So constructors look a lot like methods, but with the containing type name in place of the usual return type and method name.

Example 3-14. A class with one constructor

```
public class Item
{
    public Item(decimal price, string name)
    {
        _price = price;
        _name = name;
    }
    private readonly decimal _price;
    private readonly string _name;
}
```

This constructor is pretty simple: it just copies its arguments to fields. A lot of constructors do no more than that. You're free to put as much code in there as you like, but by convention, developers usually expect the constructor not to do very much—its main job is to ensure that the object is in a valid initial state. That might involve checking the arguments and throwing an exception if there's a problem, but not much else. You are likely to surprise developers who use your class if you write a constructor that does something non-trival, such as adding data to a database or sending a message over the network.

Example 3-15 shows how to use a constructor that takes arguments. We just use the `new` operator, passing in suitably typed values as arguments.

Example 3-15. Using a constructor

```
var item1 = new Item(9.99, "Hammer");
```

You can define multiple constructors, but it needs to be possible to distinguish between them: you cannot define two constructors that both take the same number of arguments of the same types, because there's no way for the `new` keyword to choose between them.

If you do not define any constructors at all, C# will provide a *default constructor* that is equivalent to an empty constructor that takes no arguments. (And as mentioned earlier, if you're writing a struct, you'll get that even if you do define other constructors.)

[4] There's an exception. If a class supports a CLR feature called serialization, objects of that type can be deserialized directly from a data stream, bypassing constructors. But even here, you can dictate what data is required.

> Although the C# specification unambiguously defines a default constructor as one generated for you by the compiler, be aware that there's another widely used meaning. Some of Microsoft's documentation uses the term "default constructor" to mean any public, parameterless constructor, regardless of whether it was generated by the compiler. There's some logic to this—from the perspective of some code using a class, it's not possible to tell the difference between a compiler-generated constructor, and an explicit zero-argument constructor, so if the term default constructor is to mean anything useful from that perspective, it can only mean a public constructor that takes no arguments. However, that's not how the C# specification defines the term.

The compiler-generated default constructor does nothing beyond the zero-initialization of fields that occurs for all objects. However, there are some situations in which it is necessary to write your own parameterless constructor. You might need the constructor to execute some code. Example 3-16 sets an `_id` field based on a static field that it increments for each new object, to give each instance a distinct ID. This doesn't require any arguments to be passed in, but it does involve running some code. (You couldn't do this in a struct of course, because their no-arguments constructors are always the compiler-generated ones that do nothing more than zeroing all the fields.)

Example 3-16. A non-empty zero-argument constructor

```
public class ItemWithId
{
    private static int _lastId;
    private int _id;

    public ItemWithId()
    {
        _id = ++_lastId;
    }
}
```

There is another way to achieve the same effect as Example 3-16. I could have written a static method called `GetNextId`, and then used that in the `_id` field initializer. Then I wouldn't have needed to write this constructor. However, there is one advantage to the approach in Example 3-16: it turns out that field initializers are not allowed to invoke non-static methods. That's because the object is in an incomplete state during field initialization, so it may be dangerous to call non-static methods—they may rely on fields having valid values. But an object is allowed to call its own non-static methods inside a constructor. The object's still not fully built yet of course, but it's closer to completion, and so the dangers are reduced.

There's a second reason for writing your own zero-argument constructor. If you define at least one constructor for a class, this will disable the default constructor generation. If you need your class to provide parameterized construction, but you still want to offer a no-arguments constructor, you'll need to write one, even if it's empty.

> Some frameworks can only use classes that provide a zero-argument constructor. If you build a user interface (UI) with WPF, classes that can act as custom UI elements usually need such a constructor.

If you write a type that offers a lot of constructors, you may find that they have a certain amount in common—you may need to perform some common initialization tasks. The class in Example 3-16 calculates a numeric identifier for each object in its constructor, and if it were to provide multiple constructors, they might all need to do that same work. Moving the work into a field initializer would be one way to solve that, but what if only some of the constructors wanted to do it? You might have work that was common to most constructors, but you might want to make an exception by having one constructor which allows the ID to be specified rather than calculated. The field initializer approach would no longer be appropriate because you'd want individual constructors to be able to opt in or out. Example 3-17 shows a modified version of the code from Example 3-16, defining two extra constructors.

Example 3-17. Optional chaining of constructors

```
public class ItemWithId
{
    private static int _lastId;
    private int _id;
    private string _name;

    public ItemWithId()
    {
        _id = ++_lastId;
    }

    public ItemWithId(string name)
        : this()
    {
        _name = name;
    }
    public ItemWithId(string name, int id)
    {
        _name = name;
        _id = id;
    }

}
```

If you look at the second constructor in Example 3-17, its parameter list is followed by a colon, and then `this()`, which invokes the first constructor. You can invoke any constructor that way—Example 3-17 passes no arguments, so it's invoking the no-arguments constructor. Example 3-18 shows a different way to structure all three constructors, illustrating how to pass arguments.

Example 3-18. Chained constructor arguments

```
public ItemWithId()
    : this(null)
{
}

public ItemWithId(string name)
    : this(name, ++_lastId)
{
}

private ItemWithId(string name, int id)
```

```
{
    _name = name;
    _id = id;
}
```

The two-argument constructor here is now a sort of master constructor—it's the only one that actually does any work. The other constructors just pick suitable arguments for that main constructor. Arguably this is a cleaner solution than the previous examples, because the work of initializing the fields is done in just one place, rather than having different constructors each perform their own smattering of field initialization.

Notice that I've made the two-argument constructor in Example 3-18 `private`. At first glance it can look a bit odd to define a way of building an instance of a class, and then to make it inaccessible, but it makes perfect sense when chaining constructors. And there are other scenarios in which a private constructor might be useful—we might want to write a method that makes a clone of an existing `ItemWithId`, in which case that constructor would be useful, but we probably want to keep it private so that we can retain control of exactly how new objects get created.

The constructors we've looked at so far run when a new instance of an object is created. Classes and structs can also define a static constructor. This runs at most once in the lifetime of the application. You do not invoke it explicitly—C# ensures that it runs automatically at some point before you first use the class. So unlike an instance constructor, there's no opportunity to pass arguments. Since static constructors cannot take arguments, there can be only one per class. Also, because these are never accessed explicitly, you do not declare any kind of accessibility for a static constructor. Example 3-19 shows a class with a static constructor.

Example 3-19. Class with static constructor

```
public class Bar
{
    private static DateTime _firstUsed;
    static Bar()
    {
        Console.WriteLine("Bar's static constructor");
        _firstUsed = DateTime.Now;
    }
}
```

Just as an instance constructor puts the instance into a useful initial state, the static constructor provides an opportunity to initialize any static fields.

By the way, you're not obliged to ensure that a constructor (static or instance) initializes every field. When a new instance of a class is created, the instance fields are initially all set to 0 (or the equivalent, such as `false` or `null`). Likewise, a type's static fields are all zeroed out before the class is first used. Unlike with local variables, you only need to initialize fields if you want to set them to something other than the default zero-like value.

Even then, you may not need a constructor. A field initializer may be sufficient. However, it's useful to know exactly when constructors and field initializers run. I mentioned earlier that the behavior varies according to whether constructors are present, so now that we've looked at constructors in a bit more detail, we can finally look at the whole picture of initialization.

At runtime, a type's static fields will first be set to zero (or equivalent values). Next, the field initializers are run in the order in which they are written in the source file. This ordering matters if one field's initializer refers to another. In Example 3-20, fields `a` and `b` both have the same initializer expression but they end up with different values (1 and 42 respectively) due to the order in which initializers run.

Example 3-20. Significant ordering of static fields

```
private static int a = b + 1;
private static int b = 41;
private static int c = b + 1;
```

The exact moment at which static field initializers run depends on whether there's a static constructor. As mentioned earlier, if there isn't, then the exact moment is not defined—C# guarantees to run them no later than the first access to one of the type's fields, but it reserves the right to run them arbitrarily early. But the presence of a static constructor changes matters: in that case the static field initializers run immediately before the constructor. So when does the constructor run? It will be triggered by one of two events, whichever occurs first: creating an instance, or accessing any static member of the class.

For non-static fields, the story is similar: the fields are first all initialized to zero (or equivalent values) and then field initializers run in the order in which they appear in the source file, and this happens before the constructor runs. Of course, the difference is that instance constructors are invoked explicitly, so it's clear when they will run.

I've written a class whose purpose is to examine this construction behavior, shown in Example 3-21. The class, called `InitializationTestClass`, has both `static` and non-`static` fields, all of which call a method, `GetValue` in their initializers. That method always returns the same value, 1, but it prints out a message so we can see when it is called. The method also defines a no-arguments instance constructor and a static constructor, both of which print out messages.

Example 3-21. Initialization order

```
public class InitializationTestClass
{
    public InitializationTestClass()
    {
        Console.WriteLine("Constructor");
    }

    static InitializationTestClass()
    {
        Console.WriteLine("Static constructor");
    }

    public static int s1 = GetValue("Static field 1");
    public int ns1 = GetValue("Non-static field 1");
    public static int s2 = GetValue("Static field 2");
    public int ns2 = GetValue("Non-static field 2");

    private static int GetValue(string message)
    {
        Console.WriteLine(message);
        return 1;
    }
```

```
    public static void Foo()
    {
        Console.WriteLine("Static method");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Main");
        InitializationTestClass.Foo();
        Console.WriteLine("Construct 1");
        InitializationTestClass i = new InitializationTestClass();
        Console.WriteLine("Construct 2");
        i = new InitializationTestClass();
    }
}
```

The `Main` method prints out a message, calls a static method defined by `InitializationTestClass`, and then constructs a couple of instances. Running the program, I see the following output:

```
Main
Static field 1
Static field 2
Static constructor
Static method
Construct 1
Non-static field 1
Non-static field 2
Constructor
Construct 2
Non-static field 1
Non-static field 2
Constructor
```

Notice that both static field initializers and the static constructor run before the call to the static method begins. The field initializers run before the static constructor, and as expected, they have run in the order in which they appear in the source file. Because this class includes a static constructor, we know when static initialization will begin—it is triggered by the first use of that type, which in this example will be when our `Main` method calls `InitializationTestClass.Foo`. You can see that it happened immediately before that point and no earlier, because our `Main` method managed to print out its first message before the static initialization occurred. If this example did not have a static constructor, and only had static field initializers, there would be no guarantee that that static initialization would happen at the exact same point—the specification would allow the initialization to happen earlier.

You need to be careful about what you do in code that runs during static initialization: it may get to run earlier than you expect. For example, suppose your program uses some sort of diagnostic logging mechanism, and that you need to configure this when the program starts in order to enable logging of messages to the proper location. There's always a possibility that code that runs during static initialization could execute before you've managed to do this, meaning that diagnostic logging will not yet be working correctly. That might make problems in this code hard to debug. Even when you narrow

down C#'s options by supplying a static constructor, it's relatively easy to trigger that earlier than you meant too. Any use of any static member of a class will trigger its initialization, and you can find yourself in a situation where your static constructor is kicked off by static field initializers in some other class that doesn't have a static constructor—this could happen before your `Main` method even starts.

You could try to fix this by initializing the logging code in its own static initialization. Because C# guarantees to run initialization before the first use of a type, you might think that this would ensure that the logging initialization would complete before the static initialization of any code that used the logging system. However, there's a potential problem: C# only guarantees when it will *start* static initialization for any particular class. It doesn't guarantee to wait for it to finish. It cannot make such a guarantee, because if it did, code such as that in Example 3-22 would put it in an impossible situation.

Example 3-22. Circular static dependencies

```
public class AfterYou
{
    static AfterYou()
    {
        Console.WriteLine("AfterYou static constructor starting");
        Console.WriteLine("NoAfterYou.Value: " + NoAfterYou.Value);
        Console.WriteLine("AfterYou static constructor ending");
    }

    public static int Value = 42;
}

public class NoAfterYou
{
    static NoAfterYou()
    {
        Console.WriteLine("NoAfterYou static constructor starting");
        Console.WriteLine("AfterYou.Value: " + AfterYou.Value);
        Console.WriteLine("NoAfterYou static constructor ending");
    }

    public static int Value = 42;
}
```

There is a circular relationship between the two types in this example: both have static constructors that attempt to use a static field defined by the other class. The exact behavior will depend on which of these two classes the program tries to use first. If the first to be used is `AfterYou`, I see the following output:

```
AfterYou static constructor starting
NoAfterYou static constructor starting
AfterYou.Value: 42
NoAfterYou static constructor ending
NoAfterYou.Value: 42
AfterYou static constructor ending
```

As you'd expect, the static constructor for `AfterYou` runs first, because that's the class my program is trying to use. It prints out its first message, but then it tries to use the `NoAfterYou.Value` field. That means the static initialization for `NoAfterYou` now has to start, so we see the first message from its static constructor. It then goes on to retrieve the `AfterYou.Value` field, even though the `AfterYou` static constructor

hasn't finished yet. That's OK because the ordering rules only say when static initialization is triggered, and they do not guarantee when it will finish. If they tried to guarantee complete initialization, this code would be unable to proceed—the `NoAfterYou` static constructor could not move forward because the `AfterYou` static construction is not yet complete, but that can't move forward because it would be waiting for the `NoAfterYou` static initialization to finish.

The moral of this story is that you should not get too ambitious about what you try to achieve during static initialization. It's hard to predict the exact order in which things will happen.

Methods

Methods are named bits of code that can optionally return a result, and which may take arguments. C# makes the fairly common distinction between parameters and arguments: a method defines a list of the inputs it expects, the parameters, and the code inside the method refers to these parameters by name. The values seen by the code could be different each time the method is invoked, and 'argument' refers to the specific value supplied for a parameter in a particular invocation.

As you've already seen, when an accessibility specifier such as `public` or `private` is present, this appears at the start of the method declaration. The optional `static` keyword comes next where present. After that, the method declaration states the return type. As with many C family languages, methods are not required to return anything, and you indicate this by putting the `void` keyword in place of the return type. Inside the method, you use the `return` keyword followed by an expression to specify the value for the method to return. In the case of a `void` method, you can use the `return` keyword without an expression to terminate the method, although this is optional because a `void` method will return when execution reaches the end of the method.

C# supports only a single return type. However, it is possible to have a method return multiple values, because you can designate parameters as being for output rather than input. Example 3-23 returns two values, both produced by integer division. The main return value is the quotient, but it also returns the remainder through its final parameter, which has been annotated with the `out` keyword.

Example 3-23. Returning multiple values with out

```
public static int Divide(int x, int y, out int remainder)
{
    remainder = x % y;
    return x / y;
}
```

When invoking a method of this kind, we are required to indicate explicitly that we are aware of how the method uses the argument—we must use the `out` keyword at the call site too, as Example 3-24 shows. (Some C family languages do not make any visual distinction between calls that pass values and ones that pass references, but the semantics are very different, so C# makes it explicit.)

Example 3-24. Calling a method with an out parameter

```
int r;
int q = Divide(10, 3, out r);
```

This works by passing a reference to the `r` variable, so when the `Divide` method assigns a value into `remainder`, it's really assigning it into the caller's `r` variable. This is an `int`, which is a value type, so it would not normally be passed by reference, and this kind of reference is limited. Only method arguments can use this feature. You cannot declare a local variable or field that holds such a reference, because the reference is only valid for the duration of the call. (A C# implementation could choose to implement this by putting the `r` variable in Example 3-24 on the stack and then passing a pointer to that stack location into the `Divide` method. That's workable because the reference is only required to remain valid until the method returns.)

An `out` reference requires information to flow from the method back to the caller: if the method returns without assigning something into all of its `out` arguments, you'll get a compiler error. (This requirement does not apply if the method throws an exception instead of returning.) There's a related keyword, `ref`, which has similar reference semantics, but which allows bidirectional flow of information. With a `ref` argument, it's as though the method has direct access to the variable the caller passed in—we can read its current value, as well as modifying it. (The caller is obliged to ensure that any variables passed with `ref` contain a value before making the call, so in this case, the method is not required to modify the values.) If you call a method with a parameter annotated with `ref` instead of `out`, you have to make it clear at the call site that you meant to pass a reference to a variable as the argument, as Example 3-25 shows.

Example 3-25. Calling a ref method

```
long x = 41;
Interlocked.Increment(ref x);
```

By the way, you can use the `out` and `ref` keywords with reference types too. That may sound redundant, but it can be useful. It provides double indirection—the method receives a reference to a variable that holds a reference. Normally when you pass a reference-type argument to a method, that method gets access to whatever object you choose to pass it, so while the method can use members of that object, it can't replace it with a different object. But if you mark a reference type argument with `ref`, the method has access to your variable, so it could replace it with a reference to a completely different object.

By the way, constructor arguments work in the same way as normal methods, so you can use `out` and `ref` with constructors too. Also, just to be clear, the `out` or `ref` qualifiers are part of the method (or constructor) signature. The caller passes an `out` (or `ref`) argument if and only if the parameters was declared as `out` (or `ref`). You can't decide unilaterally to pass an argument by reference to a method that does not expect it.

Method arguments can be made optional by defining default values. The method in Example 3-26 specifies the values that the arguments should have if the caller doesn't supply them. This method can then be invoked with no arguments, one argument, or both arguments.

Example 3-26. A method with optional arguments

```
public void Blame(string perpetrator = "the youth of today",
    string problem = "the downfall of society")
{
     Console.WriteLine("I blame {0} for {1}.", perpetrator, problem);
}
```

Normally, when invoking a method you specify the arguments in order. However, what if you want to call the method in Example 3-26, but you only want to provide a value for the second argument, using the default value for the first? You can't just leave the first argument empty—if you tried to write `Blame( , "everything")`, you'd get a compiler error. Instead, you can specify the name of the argument you'd like to supply, using the syntax shown in Example 3-27. C# will fill in the arguments you omit with the specified default values.

Example 3-27. Specifying an argument name

```
Blame(problem: "everything");
```

> Obviously, this only works when invoking methods that define default argument values. However, you are free to specify argument names when invoking any method—sometimes it can be useful to do this even when you're not omitting any arguments, because it can make it easier to see what the arguments are for when reading the code.

It's important to understand how C# implements default argument values. When you invoke a method without providing all the arguments, as Example 3-27 does, the compiler generates code that passes a full set of arguments as normal. It effectively rewrites your code, adding back in the arguments you left out. The significance of this is that if you write a library that defines default argument values like this, you will run into problems if you ever change the defaults. Code that was compiled against the old version of the library will have copied the old defaults into the call sites, and won't pick up the new values unless it is recompiled.

So you will sometimes see an alternative mechanism used for allowing arguments to be omitted: *overloading*, which is a slightly histrionic term for the rather mundane idea that a single name or symbol can be given multiple meanings. In fact, we already saw this technique with constructors—in Example 3-18, I defined one master constructor that did the real work, and then two other constructors that called into that one. We can use the same trick with methods as Example 3-28 shows.

Example 3-28. Overloaded method

```
public void Blame(string perpetrator, string problem)
{
    Console.WriteLine("I blame {0} for {1}.", perpetrator, problem);
}

public void Blame(string perpetrator)
{
    Blame(perpetrator, "the downfall of society");
}
public void Blame()
{
    Blame("the youth of today", "the downfall of society");
}
```

In one sense, this is slightly less flexible than default argument values, because we no longer have any way to specify a value for the `problem` argument while picking up the default `perpetrator` (although it would be easy enough to solve that by just adding a method with a different name). On the other hand, method overloading offers two potential advantages: it allows you to decide on the default values at runtime if necessary, and it also provides a way to deal with `out` and `ref` arguments. Those require references

to local variables, so there's no way to define a default value. But you can always provide overloads with and without those arguments if you need to.

Extension methods

C# lets you write methods that appear to be new members of existing types. An *extension method*, as these things are called, looks like a normal static method, but with the `this` keyword added to its first parameter. You are only allowed to define extension methods as members of a static class. Example 3-29 adds a not especially useful extension method to `string`, called `Show`.

Example 3-29. An extension method

```
namespace MyApplication
{
    public static class StringExtensions
    {
        public static void Show(this string s)
        {
            System.Console.WriteLine(s);
        }
    }
}
```

I've shown the namespace declaration in this example because namespaces are significant: extension methods are only available if you've either written a `using` directive for the namespace in which the extension is defined, or the code you're writing is defined in the same namespace. In code that does neither of these things, the `string` class will look like normal, and will not acquire the `Show` method defined by Example 3-29. However, code such as Example 3-30, which is defined in the same namespace as the extension method will find that the method is available.

Example 3-30. Extension method available due to namespace declaration

```
namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            "Hello".Show();
        }
    }
}
```

The code in Example 3-31 is in a different namespace, but it also has access to the extension method thanks to a `using` directive.

Example 3-31. Extension method available due to using directive

```
using MyApplication;

namespace Other
{
    class Program
    {
        static void Main(string[] args)
        {
            "Hello".Show();
```

```
        }
    }
}
```

Extension methods are not really members of the class for which they are defined of course—the `string` class does not truly gain an extra method in these examples. It's just an illusion maintained by the C# compiler, one that it keeps up even in situations where method invocation happens implicitly. This is particularly useful with C# features that require certain methods to be available. In Chapter 2, you saw that `foreach` loops depend on a `GetEnumerator` method. Many of the LINQ features we'll look at in Chapter 10 also depend on certain methods being present, as do the asynchronous language features described in Chapter 18. In all cases, you can enable these language features for types that do not support them directly by writing suitable extension methods.

Properties

Classes and structs can define *properties*, which are really just methods in disguise. To access a property you use a syntax that looks like field access, but which ends up invoking a method. Properties can be useful for signaling intent—when something is exposed as a property, the implication is that it represents information about the object, rather than an operation the object performs, so reading a property is usually inexpensive, and should have no significant side effects. Methods on the other hand are more likely to cause an object to do something.

Of course, since properties are just a kind of method, nothing actually enforces this. You are free to write a property that takes hours to run and which makes significant changes to your application's state whenever its value is read, but that would be a pretty lousy way to design code.

Properties typically provide a pair of methods: one to get the value and one to set it. Example 3-32 shows a very common pattern: a property with get and set methods that just provide access to a field. Why not just make the field public? That's often frowned upon because it makes it possible for external code to change an object's state without the object knowing about it. It might be that in future revisions of the code, the object needs to do something every time the value changes—perhaps it needs to update the user interface. Another reason for using properties is simply that some systems require it—some UI data binding systems are only prepared to consume properties, for example. Also, some types do not support fields—later in this chapter I'll show how to define an abstract type using an *interface*, and interfaces can contain properties, but not fields.

Example 3-32. Class with simple property

```
public class HasProperty
{
    private int _x;
    public int X
    {
        get
        {
            return _x;
        }
        set
        {
            _x = value;
        }
```

```
    }
}
```

The pattern in Example 3-32 is so common that C# can write most of it for you. Example 3-33 is more or less equivalent—the compiler generates a field for us, and generates get and set methods that retrieve and modify the value just like those in Example 3-32. The only difference is that code elsewhere in the same class can't get directly at the field in Example 3-33 because the compiler hides it.

Example 3-33. Automatic property

```
public class HasProperty
{
    public int X { get; set; }
}
```

In either case, this is just a fancy syntax for a pair of methods. The get method returns a value of the property's declared type, an `int` in this case, while the setter takes a single argument of that type through an implicit parameter called `value`—Example 3-32 makes use of that argument to update the field. You're not obliged to store the value in a field of course. In fact, nothing even forces you to make the get and set methods related in any way—you could write a getter than returns random values, and a setter that completely ignores the value you supply. However, just because you can doesn't mean you should. In practice, anyone using your class will expect properties to remember the values they've been given, not least because in use, properties look just like fields, as Example 3-34 shows.

Example 3-34. Using a property

```
var o = new HasProperty();
o.X = 123;
o.X += 432;
Console.WriteLine(o.X);
```

If you're using the full syntax to implement a property, shown in Example 3-32, you can leave out either the `set` or the `get` to make a read-only or write-only property respectively. Read-only properties can be useful for aspects of an object that are fixed for its lifetime such as an identifier. Write-only properties are less useful, although they can crop up in dependency injection systems. You can't make a read-only or write-only property with the automatic property syntax shown in Example 3-33, because you wouldn't be able to do anything useful with the property. However, you might want to define a property where the getter is public but the setter is not. You can do this with either the full or the automatic syntax. Example 3-35 shows how this looks with the latter.

Example 3-35. Automatic property with private setter

```
public int X { get; private set; }
```

Speaking of read-only properties, there's an important issue to be aware of involving properties, value types, and immutability.

Properties and mutable value types

As I mentioned earlier, most value types are immutable, but it's not a requirement. Modifiable types can cause a problem if you use them with properties. In general, one of the issues with mutable value types is that you can end up accidentally modifying a copy of the value rather than the one you meant, and this issue becomes apparent if you define

a property that uses a mutable value type. The `Point` struct in the `System.Windows` namespace is modifiable, so we can use it to illustrate the problem. Example 3-36 defines a `Location` property of this type.

Example 3-36. A property using a mutable value type

```
using System.Windows;

public class Item
{
    public Point Location { get; set; }
}
```

The `Point` type defines read/write properties called `X` and `Y`, so given a variable of type `Point`, you can set these properties. However, if you try to set either of these properties via another property, the code will not compile. Example 3-37 does this—it attempts to modify the `X` property of a `Point` retrieved from an `Item` object's `Location` property.

Example 3-37. Error: cannot modify a property of a value type property

```
var item = new Item();
item.Location.X = 123;
```

This example produces the following error:

```
error CS1612: Cannot modify the return value of 'Item.Location' because it is
not a variable
```

C# considers fields to be variables as well as local variables and method arguments, so if we were to modify Example 3-36 so that `Location` was a public field rather than a property, Example 3-37 would then compile, and would work as expected. But why doesn't it work with a property? Remember that properties are just methods, so Example 3-36 is more or less equivalent to Example 3-38.

Example 3-38. Replacing a property with methods

```
using System.Windows;

public class Item
{
    private Point _location;
    public Point get_Location()
    {
        return _location;
    }
    public void set_Location(Point value)
    {
        _location = value;
    }
}
```

Since `Point` is a value type, `get_Location` has to return a copy—there's no way it can return a reference to the value in the `_location` field. Since properties are methods in disguise, Example 3-36 also has to return a copy of the property value, so if the compiler did allow Example 3-37 to compile, we would be setting the `X` property on the copy returned by the property, and not the actual value in the `Item` object that the property represents. Example 3-39 makes this explicit, and it will in fact compile—the compiler will let us shoot ourselves in the foot if we make it sufficiently clear that we

really want to. And with this version of the code, it's quite clear that this will not modify the value in the `Item` object.

Example 3-39. Making the copy explicit

```
var item = new Item();
Point location = item.Location
location.X = 123;
```

So why does it work if we use a field instead of a property? The clue is in the compiler error: if we want to modify a struct instance, we must do so through a variable. In C# a variable is a storage location, and when we refer to a particular field by name, it's clear that we want to work with the storage location that holds that field's value. But methods (and therefore properties) cannot return something that represents a storage location—they can only return the value that is in a storage location. In other words, C# has no equivalent of `ref` for return values. Fortunately, most value types are immutable, and this problem only arises with mutable value types. Avoid those, and you won't run into this problem.

Since properties are really just methods (typically in pairs), in theory they could accept arguments beyond the implicit `value` argument used by `set` methods. The CLR allows this but C# does not support it except for one special kind of property: an indexer.

Indexers

An indexer is a property that takes one or more arguments, and which is accessed with the same syntax as is used for arrays. This is useful when writing a class that contains a collection of objects. Example 3-40 uses one of the collection classes provided by the .NET Framework. It is essentially a variable-length array, and it's able to feel like a native array thanks to its indexer, used on the 2nd and 3rd lines. (I'll describe arrays in detail in Chapter 5.)

Example 3-40. Using an indexer

```
var numbers = new List<int> { 1, 2, 1, 4 };
numbers[2] += numbers[1];
Console.WriteLine(numbers[0]);
```

From the CLR's point of view, an indexer is a property much like any other, except that it has been designated as the *default property*. This concept is something of a hangover from the old COM-based versions of Visual Basic that got carried over into .NET, and which C# mostly ignores. Indexers are the only C# feature that treats default properties as being special. If a class designates a property as being the default one, and if the property accepts at least one argument, C# will let you use that property through the indexer syntax.

The syntax for declaring indexers is somewhat idiosyncratic. Example 3-41 shows a read-only indexer. You could add a `set` method to make it read/write, just like with any other property. (Incidentally, all properties have names, including the default one. C# calls the indexer property `Item`, and automatically adds the annotation indicating that it's the default property. You won't normally refer to an indexer by name, but the name will be visible in some tools. A lot of the classes in the .NET Framework list their indexer under the name `Item` in the documentation.)

Example 3-41. Class with indexer

```
public class Indexed
{
    public string this[int index]
    {
        get
        {
            return index < 5 ? "Foo" : "bar";
        }
    }
}
```

There is some logic to this syntax. The CLR allows any property to accept arguments, so in principle, any property could be indexed. So you could imagine a property declaration of the form shown in Example 3-42. If that were the supported pattern, then it would make some sense to use the `this` keyword in place of the property name when declaring the default property.

Example 3-42. A hypothetical named, indexed property

```
public string X[int index]  // Will not compile!
{
    get ...
}
```

As it happens, C# doesn't support that more generalized syntax—only the default property can be indexed. I only show Example 3-42 because it makes the supported indexer syntax seem slightly less peculiar.

C# supports multi-dimensional indexers. These are simply indexers with more than one parameter—since properties are really just methods, you can define indexers with any number of parameters.

Operators

Classes and structs can define customized meanings for operators. This is sometimes referred to as overloading, although it's slightly different from method overload, which allows a single class to define multiple methods with the same name. With operators, overloading just means that the operator has different meanings in different contexts; any single class only gets to define one meaning for any particular operator.

I already showed some custom operators—Example 3-7 defined a custom implementation for the `++` operator, and Example 3-11 implemented `==` and `!=`. You can define custom implementations for almost all of the arithmetic, logical, and relational operators introduced in Chapter 2. Of the operators shown in Tables 2-3, 2-4, 2-5, and 2-6, you can define custom meanings for all except the conditional AND (`&&`) and conditional OR (`||`) operators. Those operators are evaluated in terms of other operators however, so by defining logical AND (`&`), logical OR (`|`) and also the logical `true` and `false` operators (described shortly) you can control the way that `&&` and `||` work for your type even though you cannot implement them directly.

All custom operator implementations follow a certain pattern. They look like static methods, but in the place where you'd normally expect the method name, you instead have the `operator` keyword followed by the operator for which you want to define a custom meaning. This is followed by a parameter list, and the number of parameters is determined by the number of operands the operator requires. Example 3-7 showed an

operator with a single parameter, the unary `++` operator. Example 3-43 shows how the binary `+` operator would look for the same class.

Example 3-43. Implementing the + operator

```
public static Counter operator +(Counter x, Counter y)
{
    return new Counter(x.Count + y.Count);
}
```

C# requires certain operators to be defined in pairs. We already saw this with the `==` and `!=` operator—it is illegal to define one and not the other. Likewise, if you define the `>` operator for your type, you must also define the `<` operator, and vice versa. The third and final such pair is `>=` and `<=`.

When you overload an operator for which a compound assignment operator exists, you are in effect defining behavior for both. If you define custom behavior for the `+` operator, the `+=` operator will automatically work too, for example.

The `operator` keyword can also define custom conversions, methods that convert your custom type to some other type, or vice versa. For example, if we wanted to be able to convert `Counter` objects to and from `int`, we could add the two methods in Example 3-44 to the class.

Example 3-44. Conversion operators

```
public static explicit operator int(Counter value)
{
    return value.Count;
}

public static explicit operator Counter(int value)
{
    return new Counter(value);
}
```

I've used the `explicit` keyword here, which means that these conversions are accessed with the cast syntax, as Example 3-45 shows.

Example 3-45. Using explicit conversion operators

```
var c = (Counter) 123;
var v = (int) c;
```

If you use the `implicit` keyword instead of `explicit`, your conversion will be able to happen without needing a cast. In Chapter 2 we saw that in certain situations, C# will automatically promote numeric types. For example, you can use an `int` where a `long` is expected, perhaps as an argument for a method or in an assignment. Conversion from `int` to `long` will always succeed, and can never lose information, so the compiler will automatically generate code to perform the conversion without requiring an explicit cast. If you write `implicit` conversion operators, the C# compiler will silently use them in exactly the same way, enabling your custom type to be used in places where some other type was expected. (In fact, the C# specification defines numeric promotions such as conversion from `int` to `long` as built-in implicit conversions.)

Implicit conversion operators are something you shouldn't need to write very often. You should only do so when you can meet the same standards as built-in promotions: the conversion must always be possible and should never throw an exception. Moreover, the

conversion should make sense—`implicit` conversions are a little sneaky in that they allow you to cause methods to be invoked in code that doesn't look like it's calling a method. So unless you're intending to confuse other developers, you should only write implicit conversions where they seem to make unequivocal sense.

C# recognizes two more operators: `true` and `false`. These are a bit of an oddball pair, because although the C# specification defines them as unary operator overloads, they don't correspond directly to any operator you can write in an expression. They come into play when evaluating expressions which use your custom type as the operands of a conditional Boolean operator (either `&&` or `||`). Remember that these operators will only evaluate their second operand if the first outcome does not fully determine the result. To customize the behavior of these operators, you must define the non-conditional versions of the operators (`&` and `|`), and you must also define the `true` and `false` operators. When evaluating `&&`, C# will use your `false` operator on the first operand, and if that indicates that the first operand is false, then it will not bother to evaluate the second operand. If the first operand is not false, it will evaluate the second operand and then pass both into your custom `&` operator. The `||` operator works in much the same way, but with the `true` and `|` operators respectively.

You may be wondering why we need special `true` and `false` operators—couldn't we just define an implicit conversion to the `bool` type? In fact we can, and if we do that instead of providing `&`, `|`, `true`, and `false`, C# will use that to implement `&&` and `||` for our type. However, some types may want to represent values that are neither true nor false—there may be a third value representing an unknown state. The `true` operator allows C# to ask the question "is this definitely true?" and for the object to be able to answer "no" without implying that it's definitely false. A conversion to `bool` does not support that.

No other operators can be overloaded. For example, you cannot define custom meanings for the `.` operator used to access members of a method, or the conditional (`? :`), the null coalescing (`??`) or the `new` operators.

Events

Structs and classes can declare *events*. This is a kind of member that enables a type to provide notifications when interesting things happen, using a subscription-based model. For example, a UI object representing a button might define a `Click` event, and you can write code that subscribes to that event.

Events depend on delegates, and since Chapter 9 is dedicated to these topics, I won't go into any detail here. I'm only mentioning them because this section on type members would otherwise be incomplete.

Nested Types

The final kind of member we can define in a class or a struct is a nested type. You can define nested classes, structs, or any of the other types described later in this chapter. A nested type can do anything its normal counterpart would do, but it gets a couple of additional features.

When a type is nested, you have more choices for accessibility. A type defined at global scope can only be `public` or `internal`—`private` would make no sense because we

use it to ensure that something is only accessible from within its containing type, and there is no containing type when you define something at global scope. But a nested type does have a containing type, so if you define a nested type and make it `private`, that type can only be used from inside the type within which it is nested. Example 3-46 shows a private class.

Example 3-46. A private nested class

```
private class Program
{
    private static void Main(string[] args)
    {
        // Ask the class library where the user's My Documents folder lives
        string path =
            Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
        string[] files = Directory.GetFiles(path);
        var comparer = new LengthComparer();
        Array.Sort(files, comparer);
        foreach (string file in files)
        {
            Console.WriteLine(file);
        }
    }

    private class LengthComparer : IComparer<string>
    {
        public int Compare(string x, string y)
        {
            int diff = x.Length - y.Length;
            return diff == 0 ? x.CompareTo(y) : diff;
        }
    }
}
```

Private classes can be useful in scenarios like this where you are using an API that requires an implementation of a particular interface. In this case, I'm calling `Array.Sort` to sort a list of filenames by length. (This is not useful, but it looks nice.) I'm providing the custom sort order in the form of an object that implements `IComparer<string>`. I'll describe interfaces in detail in the next section, but this interface is just a description of what the `Array.Sort` method needs us to provide. I've written a custom class to implement this interface. This class is just an implementation detail of the rest of my code, so I really don't want to make it public. A nested private class is just what I need.

Code in a nested type is allowed to use non-public members of its containing type. However, that's a purely static relationship. An instance of a nested type does not automatically get a reference to an instance of its containing type. (If you're familiar with Java, this may surprise you. C# nested classes are equivalent to Java static nested classes, and there is no equivalent to an inner class.) So if you need nested instances to have a reference to their container, you will need to declare a field to hold that, and arrange for it to be initialized—this would work in exactly the same way as any object that wants to hold a reference to another object. Obviously, it's only an option if the outer type is a reference type.

So far, we've only looked at classes and structs, but there are some other ways to define custom types in C#. Some of these are complicated enough to warrant getting their own chapters, but there are a couple of simpler ones that I'll discuss here.

Interfaces

An interface defines a programming interface, but is entirely devoid of implementation. Classes can choose to provide an implementation of an interface. You can write code that works in terms of an interface, meaning it will be able to work with anything that implements that interface, instead of being limited to working with one particular type.

For example, the .NET Framework defines an interface called `IEnumerable<T>`, which defines a minimal set of members for representing sequences of values. (It's a generic interface, so it can represent sequences of anything. An `IEnumerable<string>` is a sequence of strings, for example. Generic types are discussed in Chapter 4.) If a method has a parameter of type `IEnumerable<string>`, you can pass it a reference to an instance of any type that implements the interface, which means that a single method can work with arrays, various collection classes provided by the .NET Framework Class Library, certain LINQ features, and many more things besides.

An interface declares methods, properties, and events, but it does not define their contents, as Example 3-47 shows. Properties indicate whether getters and/or setters should be present, but we have a semicolon in place of the body because there's no implementation. An interface is effectively a list of the members that a type will need to provide if it wants to implement the interface.

Example 3-47. An interface

```
public interface IDoStuff
{
    string this[int i] { get; set; }
    string Name { get; set; }
    int Id { get; }
    int SomeMethod(string arg);
    event RoutedEvent Click;
}
```

The individual members are not allowed accessibility modifiers—accessibility is controlled at the level of the interface itself. (Like classes, interfaces are either `public` or `internal`, unless they are nested, in which case they can have any accessibility.) Interfaces cannot contain fields or nested types because interfaces only define the API, not the implementation. Also, interfaces cannot declare constructors—an interface only gets to say what services an object should supply once it has been constructed.

By the way, most interfaces in .NET follow the convention that their name starts with an uppercase `I` followed by one or more words in PascalCase form.

A class declares the interfaces that it implements in a list after a colon following the class name as Example 3-48 shows. It should provide implementations of all the members listed in the interface, and you'll get a compiler error if you leave any out.

Example 3-48. Implementing an interface

```
public class DoStuff : IDoStuff
```

```
{
    public string this[int i] { get { return i.ToString(); } }
    public Name { get; set; }
    ...etc
}
```

When we implement an interface in C#, we typically define each of that interface's methods as a public member of our class. However, sometimes you may want to avoid this. Occasionally some API may require you to implement an interface which you feel pollutes the purity of your class's API. Or, more prosaically, you may already have defined a member with the same name and signature as a member required by the interface, but which does something different from what the interface requires. Or worse, you may need to implement two different interfaces, both of which define members with the same name and signature but which require different behavior. You can solve any of these problems with a technique called *explicit implementation* to define members that implement a member of a specific interface without being public. Example 3-49 shows the syntax for this, with an implementation of one of the methods from the interface in Example 3-47. With explicit implementations you do not specify the accessibility, and you prefix the member name with the interface name.

Example 3-49. Explicit implementation of an interface member

```
int IDoStuff.SomeMethod(string arg)
{
    ...
}
```

When a type uses explicit interface implementation, those members cannot be used through a reference of the type itself. They only become visible when referring to an object through a variable of the interface's type.

When a class implements an interface, it becomes implicitly convertible to that interface type. So you can pass any variable of `DoStuff` as a method argument of type `IDoStuff`, for example.

Interfaces are reference types. Despite this, you can implement interfaces on both classes and structs. However, you need to be careful when doing so with a struct, because when you get hold of an interface-typed reference to a struct, the struct will often end up being copied into a *box*, which is effectively an object that holds a copy of a struct in a way that can be referred to via a reference. We'll look at boxing in Chapter 7.

Enums

The `enum` keyword declares a very simple type that defines a set of named values. Example 3-50 shows an `enum` that define a set of mutually exclusive choices. You could say that this 'enumerates' the options, which is where the `enum` keyword gets its name.

Example 3-50. An enum with mutually exclusive options

```
public enum PorridgeTemperature
{
    TooHot,
    TooCold,
    JustRight
}
```

An `enum` can be used in most places you might use a type—it could be a local variable, a field, or a method parameter for example. But one of the most common ways to use an `enum` is in a `switch` statement, as Example 3-51 shows.

Example 3-51. Switching with an enum

```
switch (porridge.Temperature)
{
case PorridgeTemperature.TooHot:
    GoOutsideForABit();
    break;

case PorridgeTemperature.TooCold:
    MicrowaveMyBreakfast();
    break;

case PorridgeTemperature.JustRight:
    NomNomNom();
    break;
}
```

As this illustrates, to refer to enumeration members you must qualify them with the type name. In fact, an `enum` is really just a fancy way of defining a load of `const` fields. The members are all just `int` values under the covers. You can even specify the values explicitly, as Example 3-52 shows.

Example 3-52. Explicit enum values

```
[System.Flags]
public enum Ingredients
{
    Eggs = 1,
    Bacon = 2,
    Sausages = 4,
    Mushrooms = 8,
    Tomato = 0x10,
    BlackPudding = 0x20,
    BakedBeans = 0x40,
    TheFullEnglish = 0x7f
}
```

This example also shows an alternative way to use an `enum`. The options in Example 3-52 are not mutually exclusive. As a developer you should recognize most of those constant values as being nice round numbers in binary. (And just in case you've not memorized these numbers, in binary they are 1, 10, 100, 1000, etc. I've used hexadecimal literals here because they make it easier to see that these are round numbers in binary.) This makes it very easy to combine them together—`Eggs` and `Bacon` would be 3 (11 in binary) while `Eggs`, `Bacon`, `Sausages`, `BlackPudding`, and `BakedBeans` (my preferred combination) would be 103 (1100111 in binary, or 0x67 in hex).

> When combining flag-based enumeration values, we normally use the bitwise OR operator. For example, you could write `Ingredients.Eggs|Ingredients.Bacon`. Not only is this significantly easier to read than using the numeric values, it also works well with Visual Studio's search tools—you can find all the places a

particular symbol is used by right clicking on its definition and choosing Find All References from the context menu.

When you declare an `enum` that's designed to be combined in this way you're supposed to annotate it with the `Flags` custom attribute, which is defined in the `System` namespace. (Chapter 15 will describe attributes in detail.) Example 3-52 does this, although in practice, it doesn't matter if you forget because the C# compiler doesn't care, and in fact there are very few tools that pay any attention to it. The main benefit is that if you call `ToString` on an `enum` value, it will notice when the `Flags` attribute is present. For this `Ingredients` type, `ToString` would convert the value of 3 to the string `Eggs, Bacon`, whereas without the `Flags` attribute, it would treat that as an unrecognized value and would just return a string containing the digit `3`.

With this sort of flags-style enumeration, you can run out of bits fairly quickly. By default `enum` uses `int` to represent the value, and with a sequence of mutually exclusive values, that's usually sufficient. It would be a fairly complicated sequence that needed billions of different values in a single enumeration type. However, with one bit per flag, an `int` provides space for just 32 flags. Fortunately, you can get a little more breathing room, because you can specify a different underlying type—you can use any built-in integer type, meaning that you can go up to 64 bits. As Example 3-53 shows, you can specify the underlying type after a colon following the `enum` type name.

Example 3-53. 64-bit enum

```
[System.Flags]
public enum TooManyChoices : long
{
    ...
}
```

All `enum` types are value types incidentally, like the built-in numeric types, or any struct. But they are very limited. You cannot define any members other than the constant values—no methods or properties, for example.

Enumeration types can sometimes enhance the readability of code. A lot of APIs accept a `bool` to control some aspect of their behavior, but might often have done better to use an `enum`. Consider the code in Example 3-54. It constructs a `StreamReader`, a class for working with streams that contain text. The second constructor argument is a `bool`.

Example 3-54. Unhelpful use of bool

```
var rdr = new StreamReader(stream, true);
```

It's not remotely obvious what that second argument does. If you happen to be familiar with `StreamReader`, you may know that this argument determines whether byte ordering should be set explicitly from the code, or determined from a preamble at the start of the stream. (Using the named argument syntax would help here.) And if you've got a really good memory, you might even know which of those choices `true` happens to select. But most mere mortal developers will probably have to reach for the IntelliSense or even the documentation to work out what that argument does. Compare that experience with Example 3-55, which shows a different type.

Example 3-55. Clarity with an enum

```
var fs = new FileStream(path, FileMode.Append);
```

This constructor's second argument uses an enumeration type, which makes for rather less opaque code. It doesn't take an eidetic memory to work out that this code intends to append data to an existing file.

As it happens, this particular API has more than two options, so it couldn't use a `bool`. `FileMode` really had to be an `enum`. But it does illustrate that even in cases where you're selecting between just two choices, it's well worth considering defining an `enum` for the job, so that it's completely obvious which choice is being made when you look at the code.

Other Types

We're almost done with our survey of types and what goes in them. There's one kind type that I'll not discuss until Chapter 9: delegates. We use delegates when we need a reference to a function, but the details are somewhat involved.

I've also not mentioned pointers. C# supports pointers that work in a pretty similar way to C-style pointers, complete with pointer arithmetic. These are a little weird, because they are slightly outside of the rest of the type system. For example, in Chapter 2, I mentioned that a variable of type `object` can refer to "almost anything." The reason I had to qualify that is that pointers are the exception—`object` can work with any C# data type except a pointer. I'll be discussing pointers in Chapter 23.

But now we really are done. Some types in C# are special, including the intrinsic types, structs, interfaces, enums, delegates and pointers, but everything else looks like a class. There are a few classes that get special handling in certain circumstances, notably attribute classes (Chapter 15) and exception classes, (Chapter 8), but except for certain special scenarios, even those are otherwise completely normal classes. Even though we've seen all the kinds of types that C# supports, there's one way to define a class that I've not shown yet.

Anonymous Types

If you need a type that is nothing more than a handful of values stored in properties, C# can generate a suitable class for you. Example 3-56 shows how to create an instance of an *anonymous type* as such types are called, and shows how to use it.

Example 3-56. An anonymous type

```
var x = new { Title = "Lord", Surname = "Voldemort" };

Console.WriteLine("Welcome, " + x.Title + " " + x.Surname);
```

As you can see, we use the `new` keyword without specifying a type name. Instead, we just place a series of name/value pairs inside braces. The C# compiler will provide a type that has one read-only property for each entry inside the braces. So in Example 3-56, the variable `x` will refer to an object that has two properties, `Title` and `Surname`, both of type `string`. (You do not state the property types explicitly in an anonymous type. The compiler infers the type from the initialization expression in the same way as it does for the `var` keyword.) Since these are just normal properties, we can access them with the usual syntax, as the final line of the example shows.

The compiler generates a fairly ordinary class definition for each anonymous type. Rather usefully, it overrides `Equals` so that you can compare instances by value, and it also provides a matching `GetHashCode` implementation. The only unusual thing about the generated class is that it's not possible to refer to the type by name in C#. Running Example 3-56 in the debugger, I find that the compiler has chosen the name `<>f__AnonymousType0'2`. This is not a legal identifier in C# because of those angle brackets (`<>`) at the start. C# uses names like this whenever it wants to create something that is guaranteed not to collide with any identifiers you might use in your own code, or which it wants to prevent you from using directly; this sort of identifier is called, rather magnificently, an *unspeakable name*.

Because you cannot write the name of an anonymous type, you cannot return one from a method, or accept one as a value (unless you use an anonymous type as an inferred generic type argument, something we'll see in Chapter 4). So these would seem to be of limited value—they are only usable within the method that defines them. They were added to the language for LINQ's benefit: they enable a query to select specific columns or properties from some source collection, and also to define custom grouping criteria as you'll see in Chapter 10.

Partial Types and Methods

There's one last topic I want to discuss relating to types, something you will almost certainly encounter on a regular basis. C# supports what it calls a partial type declaration. This is a very simple concept: it means that the type declaration might span multiple files. If you add the `partial` keyword to a type declaration, C# will not complain if another file defines the same type—it will simply act as though all the members defined by the two files had appeared in a single declaration in one file.

This feature exists to make it easier to write code generation tools. Various features in Visual Studio can generate bits of your class for you. This is particularly common with user interfaces. UI applications typically have markup that defines the layout and content of each part of the UI, and you can choose for certain UI elements to be accessible in your code. This is usually achieved by adding a field to a class associated with the markup file. To keep things simple, all the parts of the class that Visual Studio generates go in a separate file from the parts that you write. This means that the generated parts can be rebuilt from scratch whenever needed without any risk of overwriting the code that you've written. Before partial types were introduced to C#, all the code for a class had to go in one file, and from time to time, code generation tools would get confused, leading to loss of code.

> Partial classes are not limited to code generation scenarios, so you can of course use this to split your own class definitions across multiple files. However, if you've written a class so large and complex that you feel the need to split it into multiple source files just to keep it manageable, that's probably a sign that the class is too complex. A better response to this problem might be to change your design.

Partial methods are also designed for code generation scenarios, but they are slightly more complex. They allow one file, typically a generated file, to declare a method, and for another file to implement the method. (Strictly speaking, the declaration and implementation are allowed to be in the same file, but they're usually not.) This may

sound like the relationship between an interface and a class that implements that interface, but it's not quite the same. With partial methods, the declaration and implementation are in the same class—they're only in different files because the class has been split across multiple files.

If you do not provide an implementation of a partial method, the compiler acts as though the method isn't there at all, and any code that invokes the method is simply ignored at compile time. The main reason for this is to support code generation mechanisms that are able to offer all sorts of notifications, but where you want zero runtime overhead for notifications that you don't need. Partial methods enable this by letting the code generator declare a partial method for each kind of notification it provides, and to generate code that invokes all of these partial methods where necessary. All code relating to notifications for which you do not write a handler method will be stripped out at compile time.

It's a slightly idiosyncratic mechanism, but it was driven by frameworks that provide extremely fine-grained notification and extension points. There are some more obvious runtime techniques you could use instead such as interfaces, or some features that I'll cover in later chapters such as callbacks or virtual methods. However, any of these would impose a relatively high cost for unused features. Unused partial methods get stripped out at compile time, reducing the cost of the bits you don't use to nothing, which is a considerable improvement.

Summary

You've now seen most of the kinds of types you can write in C#, and the sorts of members they support. Classes are the most widely used, but structs are useful if you need value-like semantics for assignment and arguments; both support the same member types, namely fields, constructors, methods, properties, indexers, events, custom operators, and nested types. Interfaces are abstract, so they only support methods, properties, indexers, and events. And enums are very limited, providing just a set of known values.

There's another feature of the C# type system that makes it possible to write very flexible types, called generic types. We'll look at these in the next chapter.

4

Generics

In Chapter 3, I showed how to write types, and described the various kinds of members they can contain. However, there's an extra dimension to classes, structs, interfaces, and methods that I did not show. They can define *type parameters*, which are placeholders into which you can plug different types at compile time. This lets you write just one type and then produce multiple versions of it. This is called a *generic type*. For example, the class library defines a generic class called `List<T>` that acts as a variable-length array. `T` is a type parameter here, and you can use any type as an argument, so `List<int>` is a list of integers, `List<string>` is a list of strings, and so on. You can also write a *generic method*, which is a method that has its own type arguments, independently of whether its containing type is generic.

Generic types and methods are visually distinctive because they always have angle brackets (`<` and `>`) after the name. These contain a comma-separated list of parameters or arguments. The same parameter/argument distinction applies here as with methods: the declaration specifies a list of parameters, and then when you come to use the method or type, you supply arguments for those parameters. So `List<T>` defines a single type parameter, `T`, and `List<int>` supplies a *type argument*, `int`, for that parameter.

Type parameters can be called whatever you like, within the usual constraints of what constitutes a legal identifier in C#. There's a common but not universal convention of using `T` when there's only one parameter. For multi-parameter generics, you tend to see slightly more descriptive names. For example, the class library defines the `Dictionary<TKey, TValue>` collection class. And sometimes you will see a descriptive name like that even when there's just one parameter, but in any case you will tend to see a `T` prefix, so that the type parameters stand out when you use them in your code.

Generic Types

Classes, structs, and interfaces can all be generic, as can delegates, which we'll be looking at in Chapter 9. Example 4-1 shows how to define a generic class. The syntax for

structs and interfaces is much the same—the type name is followed immediately by a type parameter list.

Example 4-1. Defining a generic class

```
public class NamedContainer<T>
{
    public NamedContainer(T item, string name)
    {
        Item = item;
        Name = name;
    }

    public T Item { get; private set; }
    public string Name { get; private set; }
}
```

Inside the body of the class, you can use `T` anywhere you would normally use a type name. In this case, I've used it as a constructor argument, and also as the type of the `Item` property. I could define fields of type `T` too. (In fact I have, albeit not explicitly. The automatic property syntax generates hidden fields, so my `Item` property will have an associated hidden field of type `T`.) You can also define local variables of type `T`. And you're free to use type parameters as arguments for other generic types. My `NamedContainer<T>` could declare a variable of `List<T>`, for example.

The class that Example 4-1 defines is, like any generic type, not a complete type. A generic type declaration is *unbound*, meaning that there are type parameters that must be filled in to provide a complete definition of the type. Basic questions such as how much memory a `NamedContainer<T>` instance will require cannot be answered without knowing what `T` is—the hidden field for the `Item` property would need 4 bytes if `T` were an `int`, but 16 bytes if it were a `decimal`. The CLR cannot produce executable code for a type if it does not even know how the contents will be arranged in memory. So to use this, or any other generic type, we must provide type arguments. Example 4-2 shows how. When type arguments are supplied, the result is sometimes called a *constructed type*. (Slightly confusingly, this has nothing to do with constructors, the special kind of member we looked at in Chapter 3. In fact, Example 4-2 uses those too—it invokes the constructors of a couple of constructed types.)

Example 4-2. Using a generic class

```
var a = new NamedContainer<int>(42, "The answer");
var b = new NamedContainer<int>(99, "Number of red balloons");
var c = new NamedContainer<string>("Programming C#", "Book title");
```

You can use a constructed generic type anywhere you would use a normal type. For example, you can use them as the types for method parameters and return values, properties, or fields. You can even use one as a type argument for another generic type, as Example 4-3 shows.

Example 4-3. Constructed generic types as type arguments

```
// ...where a, and b come from Example 4-2.
var namedInts = new List<NamedContainer<int>>() { a, b };
var namedNamedItem = new NamedContainer<NamedContainer<int>>(a, "Wrapped");
```

Each distinct combination of type arguments forms a distinct type. (Or in the case of a generic type with just one parameter, each different type you supply as an argument

constructs a distinct type.) This means that `NamedContainer<int>` is a different type than `NamedContainer<string>`. That's why there's no conflict in using `NamedContainer<int>` as the type argument for another `NamedContainer` as the final line of Example 4-3 does—there's no infinite recursion here.

Because each different set of type arguments produces a distinct type, there is no implied compatibility between different forms of the same generic type. You cannot assign a `NamedContainer<int>` into a variable of type `NamedContainer<string>` or *vice versa*. It makes sense that those two types are incompatible, because `int` and `string` are quite different types. But what if we used `object` as a type argument? As Chapter 2 described, you can put almost anything in an `object` variable. If you write a method with a parameter of type `object`, it's OK to pass a `string`, so you might expect a method that takes a `NamedContainer<object>` to be happy with a `NamedContainer<string>`. By default, that won't work, but some generic types can declare that it wants to support such compatibility relationships. The mechanisms that support this (called *covariance* and *contravariance*) are closely related to the type system's inheritance mechanisms. Chapter 6 is all about inheritance and type compatibility, so I shall discuss how that works with generic types in that chapter.

Because the type arguments form part of the identity of a type, it's possible to introduce multiple types with the same name as long as they have different numbers of type arguments. So you could define a generic class called say, `Operation<T>`, and then another class, `Operation<T1, T2>`, and also `Operation<T1, T2, T3>` and so on, all in the same namespace, without introducing any ambiguity. When using these types, it's clear from the number of arguments which type was meant—`Operation<int>` clearly uses the first, while `Operation<string, double>` uses the second, for example. And for the same reason, you can also have a non-generic type with the same name as a generic type. So an `Operation` class would be distinct from generic types of the same name.

My `NamedContainer<T>` example doesn't do anything to instances of its type argument, `T`—it never invokes any methods, or uses any properties or other members of `T`. All it does is accept a `T` as a constructor argument, which it stores away for later retrieval. This is also true of the generic types I've pointed out in the .NET Framework class library—I've mentioned some collection classes, which are all variations on the same theme of containing data for later retrieval. There's a reason for this: a generic class can find itself working with any type, so it can presume very little about its type arguments. However, if you want to be able to presume certain things about your type arguments, you can specify *constraints*.

Constraints

C# allows you to request that certain type arguments fulfill certain requirements. For example, suppose you want to be able to create new instances of the type on demand. Example 4-4 shows a simple class that provides deferred construction—it makes an instance available through a static property, but does not attempt to construct that instance until the first time you read the property.

Example 4-4. Creating a new instance of a parameterized type

```
// For illustration only. Consider using Lazy<T> in a real program.
public static class Deferred<T>
```

```
    where T : new()
{
    private static T _instance;

    public static T Instance
    {
        get
        {
            if (_instance == null)
            {
                _instance = new T();
            }
            return _instance;
        }
    }
}
```

> You wouldn't write a class like this in practice, because the class library offers `Lazy<T>`, which does the same job, but with more flexibility. `Lazy<T>` can work correctly in multithreaded code, which Example 4-4 will not. Example 4-4 is just to illustrate how constraints work. Don't use it!

For this class to do its job, it needs to be able to construct an instance of whatever type is supplied as the argument for `T`. The `get` accessor uses the `new` keyword, and since it passes no arguments, it clearly requires the type to provide a parameterless constructor. But not all types do, so what happens if we try to use a type without a suitable constructor as the argument for `Deferred<T>`? The compiler will reject it, because it violates a constraint that this generic type has declared for `T`. Constraints appear just before the class's opening brace, and they begin with the `where` keyword. The constraint in Example 4-4 states that `T` is required to supply a zero-argument constructor.

If that constraint had not been present, the class in Example 4-4 would not compile—you would get an error on the line that attempts to construct a new `T`. A generic type (or method) is only allowed to use features that it has specified through constraints, or which are universally available. (All types offer a `ToString` method, for example, so you can invoke that on any instance without needing to specify a constraint.)

C# offers only a very limited suite of constraints. You cannot demand a constructor that takes arguments, for example. In fact, C# supports only four kinds of constraint on a type argument: a type constraint, a reference type constraint, a value type constraint, and the `new()` constraint. We just saw that last one, so let's look at the rest.

Type Constraints

You can constrain the argument for a type parameter to be compatible with a particular type. For example, you could use this to demand that the argument type implements a particular interface. Example 4-5 shows the syntax.

Example 4-5. Using a type constraint

```
using System;
using System.Collections.Generic;

public class GenericComparer<T> : IComparer<T>
```

```
    where T : IComparable<T>
{
    public int Compare(T x, T y)
    {
        return x.CompareTo(y);
    }
}
```

I'll just explain the purpose of this example before describing how it takes advantage of a type constraint. This class provides a bridge between two styles of value comparison that you'll find in .NET. Some data types provide their own comparison logic, but at times it can be more useful for comparison to be a separate function implemented in its own class. These two styles are represented by the `IComparable<T>` and `IComparer<T>` interfaces, which are both part of the class library. (They are in the `System` and `System.Collections.Generics` namespaces respectively.) I showed `IComparer<T>` in Chapter 3—an implementation of this interface can compare two objects or values of type `T`. The interface defines a single `Compare` method that takes two arguments and returns either a negative number, 0, or a positive number if the first argument is respectively less than, equal to, or greater than the second. `IComparable<T>` is very similar, but its `CompareTo` method takes just a single argument, because with this interface, you are asking an instance to compare *itself* to some other instance.

Some of the .NET class library's collection classes require you to provide an `IComparer<T>` to support ordering operations such as sorting. They use the model in which a separate object performs the comparison because this offers two advantages over the `IComparable<T>` model. First, it enables you to use data types that don't implement `IComparable<T>`. Second, it allows you to plug in different sorting orders. (For example, suppose you want to sort some strings with a case-sensitive order. The `string` type implements `IComparable<string>`, but that provides a case-insensitive order.) So `IComparer<T>` is the more flexible model. However, what if you are using a data type that implements `IComparable<T>`, and you're perfectly happy with the order that provides. What would you do if you're working with an API that demands an `IComparer<T>`?

Actually, the answer is that you'd probably just use the .NET Framework class library feature designed for this very scenario, `Comparer<T>.Default`. If `T` implements `IComparable<T>`, that property will return an `IComparer<T>` that does precisely what you want. So in practice, you wouldn't need to write the code in Example 4-5 because the .NET Framework has already written it for you. However, it's instructive to see how you'd write your own version because it illustrates how to use a type constraint.

The line starting with the `where` keyword states that this generic class requires any argument for its type parameter `T` to implement `IComparable<T>`. Without this, the `Compare` method would not compile—it invokes the `CompareTo` method on an argument of type `T`. That method is not present on all objects, and the C# compiler only allows this because we've constrained `T` to be an implementation of an interface that does offer such a method.

Interface constraints are relatively rare. If a method needs a particular argument to implement a particular interface, you wouldn't normally need a generic type constraint. You can just use that interface as the argument's type. However, Example 4-5 can't do this. You can demonstrate this by trying Example 4-6. It won't compile.

Example 4-6. Will not compile: interface not implemented

```
public class GenericComparer<T> : IComparer<T>
{
    public int Compare(IComparable<T> x, T y)
    {
        return x.CompareTo(y);
    }
}
```

The compiler will complain that I've not implemented the `IComparer<T>` interface's `Compare` method. Example 4-6 has a `Compare` method, but its signature is wrong—that first argument should be a `T`. I could also try the correct signature without specifying the constraint, as Example 4-7 shows.

Example 4-7. Will not compile: missing constraint

```
public class GenericComparer<T> : IComparer<T>
{
    public int Compare(T x, T y)
    {
        return x.CompareTo(y);
    }
}
```

That will also fail to compile, because the compiler can't find that `CompareTo` method I'm trying to use. It's the constraint for `T` in Example 4-5 that enables the compiler to know what that method really is.

Type constraints don't have to be interfaces by the way. You can use any type. For example, you can constrain a particular argument always to derive from a particular base class. More subtly, you can also define one parameter's constraint in terms of another type parameter. Example 4-8 requires the first type argument to derive from the second, for example.

Example 4-8. Constraining one argument to derive from another

```
public class Foo<T1, T2>
    where T1 : T2
...
```

Type constraints are fairly specific—they require either a particular inheritance relationship, or the implementation of specific interfaces. However, you can define slightly less specific constraints.

Reference Type Constraints

You can constrain a type argument to be a reference type. As Example 4-9 shows, this looks similar to a type constraint. You just put the keyword `class` instead of a type name.

Example 4-9. Constraint requiring a reference type

```
public class Bar<T>
    where T : class
...
```

This constraint prevents the use of value types such as `int`, `double`, or any `struct` as the type argument. Its presence enables your code to do three things that would not

otherwise be possible. First, it means that you can write code that tests whether variables of the relevant type are `null`. If you've not constrained the type to be a reference type, there's always a possibility that it's a value type, and those can't have `null` values. The second capability is that you can use the `as` operator, which we'll look at in Chapter 6. This is really just a variation on the first feature—the `as` keyword requires a reference type because it can produce a `null` result.

The third feature that a reference type constraint enables is the ability to use the constrained parameter as the argument for some other generic type or method that has the same constraint. This is a little obscure, so it may be useful to look at an example. I often find myself writing tests that create an instance of the class I'm testing, and which also need one or more fake objects to stand in for real objects that the object under test wants to interact with. Using these stand-ins reduces the amount of code any single test has to exercise, and can make it easier to verify the behavior of the object being tested. For example, my test might need to verify that my code sends messages to a server at the right moment, but I don't want to have to run a real server during a unit test, so I provide an object that implements the same interface as the class that would transmit the message, but which won't really send the message. This is such a common pattern for a test, that it might be useful to put the code into a reusable base class, and by using generics, that class can work for any combination of the type being tested and the type being faked. Example 4-10 shows a simplified version of a kind of helper class I sometimes write in these situations.

Example 4-10. Constrained by another constraint

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;

public class TestBase<TSubject, TFake>
    where TSubject : new()
    where TFake : class
{
    public TSubject Subject { get; private set; }
    public Mock<TFake> Fake { get; private set; }

    [TestInitialize]
    public void Initialize()
    {
        Subject = new TSubject();
        Fake = new Mock<TFake>();
    }
}
```

There are various ways to build fake objects for test purposes. You could just write new classes that implement the same interface as your real objects, or you could use a library that generates them for you. One such library is called Moq (an open source project available for free from http://code.google.com/p/moq/), and that's where the `Mock<T>` class in Example 4-10 comes from. It's capable of generating a fake implementation of any interface, or of any non-sealed class. It will provide empty implementations of all members by default, and you can configure more interesting behaviors if necessary. You can also verify whether the code under test used the fake object in the way you expected.

How is that relevant to constraints? The `Mock<T>` class specifies a reference type constraint on its own type argument, `T`. This is due to the way in which it creates dynamic implementations of types at runtime—it's a technique that can only work for

reference types. That means that when I use `Mock<T>` in Example 4-10, I need to make sure that whatever type argument I pass is a reference type. But the type argument I'm using is one of my class's type parameters: `TFake`. So I don't know what type that will be—that'll be up to whoever is using my class.

For my class to compile without error, I have to ensure that I have met the constraints of any generic types that I use. I have to guarantee that `Mock<TFake>` is valid, and the only way to do that is to add a constraint on my own type that requires `TFake` to be a reference type. And that's what I've done on the 3rd line of the class definition in Example 4-10. Without that, the compiler would report errors on the two lines that refer to `Mock<TFake>`.

To put it more generally, if you want to use one of your own type parameters as the type argument for a generic that specifies a reference type constraint, you'll need to specify the same constraint on your own type parameter.

Value Type Constraints

Just as you can constrain a type argument to be a reference type, you can also constrain it to be a value type. The syntax is similar to that for a reference type constraint, but with the `struct` keyword.

Example 4-11. Constraint requiring a value type

```
public class Quux<T>
    where T : struct
...
```

Before now, we've only seen the `struct` keyword in the context of custom value types, but despite how it looks, this constraint permits any of the built-in numeric types such as `int`, as well as custom structs. That's because they all derive from the same `System.ValueType` base class.

The .NET Framework's `Nullable<T>` type imposes this constraint. Recall from Chapter 3 that `Nullable<T>` provides a wrapper for value types that allows a variable to hold either a value, or no value. (C# provides a special syntax for this type: you can put a `?` on the end of any value type name. For example, `int?` means `Nullable<int>`.) The only reason this type exists is to provide nullability for types that would not otherwise be able to hold a null value. So it only makes sense to use this with a value type—reference type variables can already be set to null without needing this wrapper. The value type constraint prevents you from using `Nullable<T>` with types for which it is unnecessary.

If you'd like to impose multiple constraints for a single type argument you can just put them in a list, as Example 4-12 shows. There are a couple of ordering restrictions: if you have a reference or value type constraint, the `class` or `struct` keyword must come first in the list. If the `new()` constraint is present, it must be last.

Example 4-12. Multiple constraints

```
public class Spong<T>
    where T : IEnumerable<T>, IDisposable, new()
...
```

When your type has multiple type parameters, you can specify multiple constraints by writing one `where` clause for each type parameter you wish to constrain. In fact we saw this earlier—Example 4-10 defines constraints for both of its parameters.

Zero-like Values

There are a few features that all types support, and which therefore do not require a constraint. This includes the set of methods defined by the `object` base class, which I'll show in Chapter 6. But there's a more basic feature which can sometimes come in useful in generic code.

Variables of any type can be initialized to a default value. As you have seen in the preceding chapters, there are some situations in which the CLR does this for us. For example, all the fields in a newly-constructed object will have a known value even if we don't write field initializers, and don't supply values in the constructor. Likewise, a new array of any type will have all of its elements initialized to a known value. The CLR does this by filling the relevant memory with zeros. The exact interpretation depends on the data type. For any of the built-in numeric types, the value will quite literally be the number zero. But for non-numeric types, it's something else. For `bool`, the default is `false`, and for a reference type, it is `null`.

Sometimes, it can be useful for generic code to be able to reset a variable back to this initial default zero-like value. But you cannot use a literal expression to do this in most situations. You cannot assign `null` into a variable whose type is specified by a type parameter, unless that parameter has been constrained to be a reference type. And you cannot assign the literal `0` into any such variable because there is no way to constrain a type argument to be a numeric type.

Instead, you can request the zero-like value for any type using the `default` keyword. (This is the same keyword we saw inside a `switch` statement in Chapter 2, but used in a completely different way. C# keeps up the C-family tradition of defining multiple, unrelated meanings for each keyword.) If you write `default(`*`SomeType`*`)` where *`SomeType`* is either a type or a type parameter, you will get the default initial value for that type: zero if it is a numeric type, and the equivalent for any other type. For example, the expression `default(int)` has value of 0, `default(bool)` is `false`, and `default(string)` is `null`. You can use this with a generic type parameter to get the default value for the corresponding type argument, as Example 4-13 shows.

Example 4-13. Getting the default (zero-like) value of a type argument

```
static void PrintDefault<T>()
{
    Console.WriteLine(default(T));
}
```

Inside a generic type or method that defines a type parameter `T`, the expression `default(T)` will produce the default, zero-like value for `T`, whatever `T` may be, without requiring any constraints. So you could use the generic method in Example 4-13 to verify that the defaults for `int`, `bool`, and `string` are the values I stated. And since I've just shown you an example, this seems like a good time to talk about generic methods.

Generic Methods

As well as generic types, C# also supports generic methods. In this case, the generic type parameter list follows the method name, and precedes the method's normal parameter list. Example 4-14 shows a method with a single type parameter. It uses that parameter as its return type, and also as the element type for an array to be passed in as the method's argument. This method returns the final element in the array, and because it's generic, it will work for any array element type.

Example 4-14. A generic method

```
public static T GetLast<T>(T[] items)
{
    return items[items.Length - 1];
}
```

> You can define generic methods inside either generic types, or non-generic types. If a generic method is a member of a generic type, all of the type parameters from the containing type are in scope inside the method, as well as the type parameters specific to the method.

Just as with a generic type, you can use a generic method by specifying its name along with its type arguments.

Example 4-15. Invoking a generic method

```
int[] values = { 1, 2, 3 };
int last = GetLast<int>(values);
```

Generic methods are very similar to generic types, but with type parameters that are only in scope within the method declaration and body. You can specify constraints in much the same way as with generic types. The constraints appear after the method's parameter list and before its body, as Example 4-16 shows.

Example 4-16. A generic method with a constraint

```
public static T MakeFake<T>()
    where T : class
{
    return new Mock<T>().Object;
}
```

There's one significant way in which generic methods differ from generic types though: you don't always need to specify a generic method's type arguments explicitly.

Type Inference

The C# compiler is often able to infer the type arguments for a generic method. So I could modify Example 4-15 by removing the type argument list from the method invocation, as Example 4-17 shows. This does not change the meaning of the code in any way.

Example 4-17. Generic method type argument inference

```
int[] values = { 1, 2, 3 };
int last = GetLast(values);
```

When presented with this sort of ordinary-looking method call, if there's no non-generic method of that name available, the compiler starts looking for suitable generic methods. If the method in Example 4-14 is in scope, the compiler then attempts to deduce the type arguments. This is a pretty simple case. The method expects an array of some type `T`, and we've passed an array of type `int`, so it's not a massive stretch to work out that this code should be treated as a call to `GetLast<int>`.

It gets more complex with more intricate cases. The C# specification has about 6 pages dedicated to the type inference algorithm, but it's all to support one goal: letting you leave out type arguments when they would be redundant. By the way, type inference is always performed at compile time, so it's based on the static type of the method arguments.

Inside Generics

If you are familiar with C++ templates, you will by now have noticed that C# generics are quite different than templates. Superficially they have some similarities, and can be used in similar ways—both are suitable for implementing collection classes, for example. However, there are some template-based techniques that simply won't work in C#, such as the code in Example 4-18.

Example 4-18. A template technique that doesn't work in C# generics

```
public static T Add<T>(T x, T y)
{
    return x + y;  // Will not compile
}
```

You can do this sort of thing in a C++ template but it will not work in C#, and you cannot fix it completely with a constraint. You could add a type constraint requiring `T` to derive from some type that defines a custom `+` operator, which would get this to compile, but it would be pretty limited—it would only work for types derived from that base type. In C++, you can write a template that will add together two items of any type that supports addition, whether that's a built-in type or a custom one. Moreover, C++ templates don't need constraints—the compiler is able to work out for itself whether a particular type will work as a template argument.

The limitations of C# generics are an upshot of how they are designed to work, so it's useful to understand the mechanism. (These limitations are not specific to Microsoft's CLR by the way. They are an inevitable result of how generics fit into the design of the CLI.)

Generic methods and types are compiled without knowing which types will be used as arguments. This is the fundamental difference between C# generics and C++ templates—in C++, the compiler gets to see every instantiation of a template. But with C#, you can instantiate generic types without access to any of the relevant source code, long after the code has been compiled. After all, Microsoft wrote the generic `List<T>` class years ago, but you could write a brand new class today, and plug that in as the type argument just fine. (You might point out that the C++ standard library's `std::vector` has been around even longer. However, the C++ compiler has access to the source file that defines the class, which is not true of C# and `List<T>`. C# only sees the compiled library.)

The upshot of this is that the C# compiler needs to have enough information to be able to generate type-safe code at the point at which it compiles generic code. Take Example 4-18. It cannot know what the + operator means here, because it would be different for different types. With the built-in numeric types, that code would need to compile to the specialized IL instructions for performing addition. If that code were in a checked context (i.e., using the `checked` keyword shown in Chapter 2), we'd already have a problem, because the code for adding integers with overflow checking uses different IL opcodes for signed and unsigned integers. But since this is a generic method we may not be dealing with the built-in numeric types at all—perhaps we are dealing with a type that defines a custom + operator, in which case the compiler would need to generate a method call. (Custom operators are just methods under the covers.) Or if the type in question turns out not to support addition, the compiler should generate an error.

There are several possible outcomes, depending on the actual types involved. That would be fine if the types were known to the compiler, but it has to compile the code for generic types and methods without knowing which types will be used as arguments.

You might argue that perhaps Microsoft could have supported some sort of tentative semi-compiled format for generic code, and in a sense, they did. When introducing generics, Microsoft modified the type system, file format, and IL instructions to allow generic code to use placeholders representing type parameters to be filled in when the type is fully constructed. So why not extend it to handle operators? Why not let the compiler generate errors at the point at which you attempt to use a generic type instead of insisting on generating errors when the generic code itself is compiled? Well it turns out to be possible to plug new sets of type arguments in at runtime—the reflection API that we'll look at in Chapter 13 lets you construct generic types. So there isn't necessarily a compiler available at the point at which an error would become apparent, because not all versions of .NET make a C# compiler available at runtime. And in any case, what should happen if a generic class was written in C# but was consumed by a completely different language, one that didn't support operator overloading, perhaps? Which language's rules should apply when it comes to working out what to do with that + operator? Should it be the language in which the generic code was written, or the language in which the type argument was written? (What if there are multiple type parameters, and for each argument you use a type written in a different language?) Or perhaps the rules should come from the language that decided to plug the type arguments into the generic type or method, but what about cases where one piece of generic code passes its arguments through to some other generic entity? Even if you could decide which of these approaches would be best, it supposes that the rules used to determine what a line of code actually means are available at runtime, a presumption that once again founders on the fact that the relevant compilers will not necessarily be installed on the machine running the code.

.NET generics solve this problem by requiring the meaning of generic code to be fully defined when the generic code is compiled, by the language in which the generic code was written. If the generic code involves using methods or other members, they must be resolved statically, i.e., the identity of those members must be determined precisely at compile time. Critically, that means compile time for the generic code itself, not for the code consuming the generic code. These requirements explain why C# generics are not as flexible as a consumer-compile-time substitution model. The payoff is that you can compile generics into libraries in binary form, and they can be used by any language, with completely predictable behavior.

Summary

The most important use case for generics back when they were first introduced was to make it possible to write type-safe collection classes. .NET did not have generics at the beginning, so the collection classes available in version 1.0 used the general-purpose `object` type. This meant you had to cast objects back to their real type every time you extracted one from a collection. It also meant that value types were not handled efficiently in collections—as we'll see in Chapter 7, referring to values through an `object` requires the generation of 'boxes' to contain the values. Generics solve these problems well. They make it possible to write collection classes such as `List<T>`, which do not require casts to be useful. Moreover, because the CLR is able to construct generic types at runtime, it can generate code optimized for whatever type a collection contains. So collection classes can handle value types such as `int` much more efficiently than before generics were introduced. We'll look at some of these collection types in the next chapter.

6

Inheritance

C# classes support the popular object-oriented code reuse mechanism known as inheritance. When you write a class, you can optionally specify a base class. Your class will derive from this, meaning that everything in the base class will be present in your class, as well as any members you add.

Classes support only single inheritance. Interfaces offer a form of multiple inheritance. Value types do not support inheritance at all. One reason for this is that value types are not normally used by reference, which removes one of the main benefits of inheritance: runtime polymorphism. Inheritance is not necessarily incompatible with value-like behavior—some languages manage it—but it often has problems. For example, assigning a value of some derived type into a variable of its base type ends up losing all of the fields that the derived type added, a problem known as *slicing*. C# sidesteps this by restricting inheritance to reference types. When you assign a variable of some derived type into a variable of a base type, you're copying a reference, not the object itself, so the object remains intact. Slicing is only an issue if the base class offers a method that clones the object, and doesn't provide a way for derived classes to extend that (or it does, but some derived class fails to extend it).

Classes specify a base class using the syntax shown in Example 6-1—the base type appears after a colon that follows the class name. This example assumes that a class called `SomeClass` has been defined elsewhere in the project.

Example 6-1. Specifying a base class

```
public class Derived : SomeClass
{
}

public class AlsoDerived : SomeClass, IDisposable
{
    public void Dispose() { }
}
```

As Example 6-1 also illustrates, if the class implements any interfaces, these are also listed after the colon. If you want to derive from a class, and you want to implement

interfaces as well, the base class must appear first, as the second class shown in Example 6-1 illustrates.

You can derive from a class which in turn derives from another class. The `MoreDerived` class in Example 6-2 derives from `Derived` which in turn derived from `Base`.

Example 6-2. Inheritance chain

```
public class Base
{
}

public class Derived : Base
{
}

public class MoreDerived : Derived
{
}
```

This means that `MoreDerived` technically has multiple base classes: it derives from both `Derived` (directly) and `Base` (indirectly, via `Derived`). This is not multiple inheritance because there is only a single chain of inheritance—any single class derives directly from at most one base class.

Since a derived class inherits everything the base class has—all its fields, methods, and other members, both public and private—an instance of the derived class can do anything an instance of the base class could do. This is the classic *is a* relationship that inheritance implies in many languages. Any instance of `MoreDerived` is a `Derived`, and also a `Base`. C#'s type system recognizes this relationship.

Inheritance and Conversions

C# provides various built-in implicit conversions. In Chapter 2, we saw the conversions for numeric types, but there are also ones for reference types. If some type `D` derives from `B` (either directly or indirectly) then a reference of type `D` can be converted implicitly to a reference of type `B`. This is follows from the fact that any instance of `D` is a `B`. (This is sometimes called the *Liskov Substitution Principle* or LSP, and it's a very common feature of object-oriented systems.) This implicit conversion enables polymorphism: any code written to work in terms of `B` will be able to work with any type derived from `B`.

Obviously there is no implicit conversion in the opposite direction—although a variable of type `B` could refer to an object of type `D`, there's no guarantee that it will. There could be any number of types derived from `B`, and a `B` variable could refer to an instance of any of them. Nevertheless, you will sometimes want to attempt to convert a reference from a base type to a derived type, an operation sometimes referred to as a *downcast*. Perhaps you know for a fact that a particular variable holds a reference of a certain type. Or perhaps you're not sure, and would like your code to provide additional services for specific types. C# offers three ways to do this.

The most obvious way to attempt a downcast is to use the cast syntax, the same syntax we use for performing non-implicit numeric conversions, as Example 6-3 shows.

Example 6-3. Feeling downcast

```
public void UseAsDerived(Base baseArg)
{
    var d = (Derived) baseArg;

    ... go on to do something with d
}
```

This conversion is not guaranteed to succeed—that's why it's not built in as an implicit conversion. If you try this when the `baseArg` argument refers to something that's not an instance of `Derived`, nor something derived from `Derived`, the conversion will fail, throwing an `InvalidCastException`.

A cast is therefore only appropriate if you're confident that the object really is of the type you expect, and you consider it to be an error if it turns out not to be. This is useful when an API accepts an object that it will later give back to you. Many asynchronous APIs do this, because in cases where you launch multiple operations concurrently, you need some way of working out which particular one finished when you get a completion notification (although as we'll see in later chapters, there are various ways to tackle that problem). Since these APIs don't know what sort of data you'll want associate with an operation, they usually just take a reference of type `object`, and you would typically use a cast to turn it back into a reference of the required type when the reference is eventually handed back to you.

Sometimes, you will not know for certain whether an object is of a particular type. In this case, you can use the `as` operator instead, which allows you to attempt a conversion without risking an exception. If the conversion fails, this operator just returns null.

Example 6-4. The as operator

```
public void MightUseAsDerived(Base b)
{
    var d = b as Derived;

    if (d != null)
    {
        ... go on to do something with d
    }
}
```

Finally, it can occasionally be useful to know whether a reference refers to an object of a particular type, without actually wanting to use any members specific to that type. For example, you might want to skip some particular piece of processing for a certain derived class. The `is` operator, shown in Example 6-5, tests whether an object is of a particular type, and returns true if it is, and false otherwise.

Example 6-5. The is operator.

```
if (!(b is WeirdType))
{
    ... do the processing that everything except WeirdType requires
}
```

When converting with a cast or the `as` operator, or when using the `is` operator, you don't necessarily need to specify the exact type. These operations will succeed as long as a reference of the object's real type could be implicitly converted to the type you're looking for. For example, given the `Base`, `Derived`, and `MoreDerived` types that

Example 6-2 defines, suppose you have a variable of type `Base` that currently contains a reference to an instance of `MoreDerived`. Obviously you could cast the reference to `MoreDerived` (and both `as` and `is` would also succeed for that type) but as you'd probably expect, converting to `Derived` would work too.

These three mechanisms also work for interfaces. When you try to convert a reference to an interface type reference, conversion will succeed if the object referred to implements the relevant interface.

Interface Inheritance

Interfaces support inheritance, but it's not quite the same as class inheritance. The syntax is similar, but as Example 6-6 shows, an interface can specify multiple base interfaces, because C# supports multiple inheritance for interfaces. The reason .NET supports this despite only offering single implementation inheritance is that most of the complications and potential ambiguities that can arise with multiple inheritance do not apply to purely abstract types.

Example 6-6. Interface inheritance

```
interface IBase1
{
    void Base1Method();
}

interface IBase2
{
    void Base2Method();
}

interface IBoth : IBase1, IBase2
{
    void Method3();
}
```

As with class inheritance, interfaces inherit all of their bases' members, so the `IBoth` interface here includes `Base1Method` and `Base2Method`, as well as its own `Method3`.

Implicit conversions exist from derived interface types to their bases. For example, a reference of type `IBoth` can be assigned to a variable of type `IBase1` and also `IBase2`.

A more subtle feature of interface inheritance is that any class that implements a derived interface is required to implement that interface's base interfaces. As Example 6-7 shows, the class only needs to state that it implements the derived interface, but the compiler will act as though `IBase1` and `IBase2` were in the interface list.

Example 6-7. Implementing a derived interface

```
public class Impl : IBoth
{
    public void Base1Method()
    {
    }
```

```
    public void Base2Method()
    {
    }

    public void Method3()
    {
    }
}
```

It may seem facile to state that implementing an interface implies implementing its bases. After all, the `Impl` in Example 6-7 has to implement `Base1Method` and `Base2Method` because the derived interface `IBoth` includes all of those. Also, any reference of type `Impl` is implicitly convertible to `IBoth`, which in turn is implicitly convertible to `IBase1` and `IBase2`. However, these facts alone do not explain all the observable behavior. The fact that the class also implements all of the base interfaces makes a difference. To see why, imagine a hypothetical world in which it is possible to implement `IBoth` but not `IBase1` and `IBase2` (despite implementing all their members).

Now in general, if a reference of some class `C` is implicitly convertible to some type `T1`, and that in turn is implicitly convertible to type `T2` this does not necessarily mean that `C` is implicitly convertible to `T2`. Implicit conversions are not transitive (in the set theory sense) so C# will not chain together multiple conversions to find a path from the available type to the required type. It will only use one. In our proposed hypothetical world, if `Impl` did implement `IBoth` without implementing its bases, it would not be possible to convert a reference of type `Impl` implicitly to `IBase1` directly. You could convert implicitly to `IBoth`, and then you could implicitly convert that `IBoth` to an `IBase1`. So you can get there, but it would require two separate steps. Back in the real world, references of type `Impl` are implicitly convertible to `IBase1`, and that's only because `IBoth` obliges it to implement `IBase1`.

There's another way in which we can see that `Impl` does implement all three interfaces. The reflection API, which is the subject of Chapter 13, provides a way to discover what interfaces a type implements. And even though the declaration of `Impl` in Example 6-7 only lists one interface, `IBoth`, the reflection API reports three interfaces. This result is indistinguishable from what you would see if `Impl` had explicitly declared its support for `IBase1` and `IBase2` as well.

Generics

If you derive from a generic class, you must you supply the type arguments it requires. You can either provide concrete types, or, if your derived class is generic, it can use its own type parameters as arguments. Example 6-8 shows both techniques, and also illustrates that when deriving from a class with multiple type parameters, you can use a mixture of techniques, specifying one type argument directly, and punting on the other.

Example 6-8. Deriving from a generic base class

```
public class GenericBase1<T>
{
    public T Item { get; set; }
}
```

```
public class GenericBase2<TKey, TValue>
{
    public TKey Key { get; set; }
    public TValue Value { get; set; }
}

public class NonGenericDerived : GenericBase1<string>
{
}

public class GenericDerived<T> : GenericBase1<T>
{
}

public class MixedDerived<T> : GenericBase2<string, T>
{
}
```

Although you are free to use any of your type parameters as type arguments for a base class, you cannot derive from a type parameter. This is slightly disappointing if you are used to languages that permit such things, but the C# language specification simply forbids it.

Covariance and Contravariance

In Chapter 4, I mentioned that generic types have special rules for type compatibility, referred to as covariance and contravariance. These rules determine whether references of certain generic types are implicitly convertible to one another when implicit conversions exist between their type arguments.

> Covariance and contravariance are applicable only to the generic type arguments of interfaces and delegates. (Delegates are described in Chapter 9.) You cannot define a covariant class or struct.

Consider the simple `Base` and `Derived` classes shown earlier in Example 6-2, and look at the method in Example 6-9, which accepts any `Base`. (It does nothing with it, but that's not relevant here—what matters is what its signature says it can use.)

Example 6-9. A method accepting any Base

```
public static void UseBase(Base b)
{
}
```

We already know that as well as accepting a reference to any `Base`, this can also accept a reference to an instance of any type derived from `Base`, such as `Derived`. Bearing that in mind, consider the method in Example 6-10.

Example 6-10. A method accepting any IEnumerable<Base>

```
public static void AllYourBase(IEnumerable<Base> bases)
{
}
```

This requires an object that implements the `IEnumerable<T>` generic interface described in Chapter 5, where `T` is `Base`. What would you expect to happen if we attempted to pass an object that did not implement `IEnumerable<Base>`, but which

did implement `IEnumerable<Derived>`? Example 6-11 does this, and it compiles just fine.

Example 6-11. Passing an IEnumerable<T> of a derived type

```
IEnumerable<Derived> derivedBases =
    new Derived[] { new Derived(), new Derived() };
AllYourBase(derivedBases);
```

Intuitively, this makes sense. The `AllYourBase` method is expecting an object that can supply a sequence of objects that are all of type `Base`. An `IEnumerable<Derived>` fits the bill because it supplies a sequence of `Derived` objects, and any `Derived` object is also a `Base`. However, what about the code in Example 6-12?

Example 6-12. A method accepting any ICollection<Base>

```
public static void AddBase(ICollection<Base> bases)
{
    bases.Add(new Base());
}
```

Recall from Chapter 5 that `ICollection<T>` derives from `IEnumerable<T>`, and it adds the ability to modify the collection in certain ways. This particular method exploits that by adding a new `Base` object to the collection. That would mean trouble for the code in Example 6-13.

Example 6-13. Error: trying to pass an ICollection<T> with a derived type

```
ICollection<Derived> derivedList = new List<Derived>();
AddBase(derivedList);  // Will not compile
```

Any code that uses the `derivedList` variable will expect every object in that list to be of type `Derived` (or something derived from it, such as the `MoreDerived` class from Example 6-2). But the `AddBase` method in Example 6-12 attempts to add a plain `Base` instance. That can't be correct, and the compiler doesn't allow it. The call to `AddBase` will produce a compiler error complaining that references of type `ICollection<Derived>` cannot be converted implicitly to references of type `ICollection<Base>`.

How does the compiler know that it's not OK to do this, while the very similar-looking conversion from `IEnumerable<Derived>` to `IEnumerable<Base>` is allowed? It's not because Example 6-12 contains code that would cause a problem by the way. You'd get the same compiler error even if the `AddBase` method were completely empty. The reason we don't get an error is that the `IEnumerable<T>` interface declares its type argument `T` as covariant. You saw the syntax for this in Chapter 5, but I didn't draw attention to it, so Example 6-14 shows the relevant part from that interface's definition again.

Example 6-14. Covariant type parameter

```
public interface IEnumerable<out T> : IEnumerable
```

That `out` keyword does the job. (Again, C# keeps up the C-family tradition of giving each keyword multiple unrelated jobs—we last saw this keyword in the context of method parameters that can return information to the caller.) Intuitively, describing the type argument `T` as 'out' makes sense, in that the `IEnumerable<T>` interface only ever

provides a `T`—it does not define any members that accept a `T`. (The interface uses this type parameter in just one place: its read-only `Current` property.)

Compare that with `ICollection<T>`. This derives from `IEnumerable<T>`, so clearly it's possible to get a `T` out of it, but it's also possible to pass a `T` into its `Add` method. So `ICollection<T>` cannot annotate its type argument with `out`. (If you were to try to write your own similar interface, the compiler would produce an error if you declared the type argument as being covariant. Rather than just taking your word for it, it checks to make sure you really can't pass a `T` in anywhere.) The compiler rejects the code in Example 6-13 because `T` is not covariant in `ICollection<T>`.

The terms covariant and contravariant come from a branch of mathematics called category theory. The parameters that behave like `IEnumerable<T>`'s `T` are called covariant (as opposed to contravariant) because implicit reference conversions for the generic type work in the same direction as conversions for the type argument: `Derived` is implicitly convertible to `Base`, and since `T` is covariant in `IEnumerable<T>`, `IEnumerable<Derived>` is implicitly convertible to `IEnumerable<Base>`.

Predictably, contravariance works the other way round, and as you've probably guessed, we denote it with the `in` keyword. It's easiest to see this in action with code that uses members of types, so Example 6-15 shows a marginally more interesting pair of classes than the earlier examples.

Example 6-15. Class hierarchy with actual members

```
public class Shape
{
    public Rect BoundingBox { get; set; }
}

public class RoundedRectangle : Shape
{
    public double CornerRadius { get; set; }
}
```

Example 6-16 defines two classes that use these shape types. Both implement `IComparer<T>`, which I introduced in Chapter 4. The `BoxAreaComparer` compares two shapes based on the area of their bounding box—whichever shape covers the larger area will be deemed the 'larger' by this comparison. The `CornerSharpnessComparer` on the other hand compares rounded rectangles by looking at how pointy their corners are.

Example 6-16. Comparing shapes

```
public class BoxAreaComparer : IComparer<Shape>
{
    public int Compare(Shape x, Shape y)
    {
        double xArea = x.BoundingBox.Width * x.BoundingBox.Height;
        double yArea = y.BoundingBox.Width * y.BoundingBox.Height;

        return Math.Sign(xArea - yArea);
    }
}

public class CornerSharpnessComparer : IComparer<RoundedRectangle>
```

```
{
    public int Compare(RoundedRectangle x, RoundedRectangle y)
    {
        // Smaller corners are sharper, so smaller radius is 'greater' for
        // the purpose of this comparison, hence the backwards subtraction.
        return Math.Sign(y.CornerRadius - x.CornerRadius);
    }
}
```

References of type `RoundedRectangle` are implicitly convertible to `Shape`, so what about `IComparer<T>`? Our `BoxAreaComparer` can compare any shapes, and declares this by implementing `IComparer<Shape>`. The comparer's type argument `T` is only ever used in the `Compare` method, and that is happy to be passed any `Shape`. It will not be fazed if we pass it a pair of `RoundedRectangle` references, so our class is a perfectly adequate `IComparer<RoundedRectangle>`. An implicit conversion from `IComparer<Shape>` to `IComparer<RoundedRectangle>` therefore makes sense, and is allowed. However, the `CornerSharpnessComparer` is more fussy. It makes use of the `CornerRadius` property, which is only available on rounded rectangles, not on any old `Shape`. Therefore, no implicit conversion exists from `IComparer<RoundedRectangle>` to `IComparer<Shape>`.

This is backwards. Implicit conversion is available between `IEnumerable<T1>` and `IEnumerable<T2>` when an implicit reference conversion from `T1` to `T2` exists. But implicit conversion between `IComparer<T1>` and `IComparer<T2>` is available when an implicit reference conversion exists in the other direction: from `T2` to `T1`. That reversed relationship is the reason this is called contravariance. Example 6-17 shows an excerpt of the definition for `IComparer<T>` showing this contravariant type parameter.

Example 6-17. Contravariant type parameter

```
public interface IComparer<in T>
```

Most generic type parameters are neither covariant nor contravariant. `ICollection<T>` cannot be variant, because it contains some members that accept a `T`, and some that return one. An `ICollection<Shape>` might contain shapes that are not `RoundedRectangles`, so you cannot pass it to a method expecting an `ICollection<RoundedRectangle>`, because such a method would expect every object it retrieves from the collection to be a rounded rectangle. Conversely, an `ICollection<RoundedRectangle>` cannot be expected to allow shapes other than rounded rectangles to be added, and so you cannot pass an `ICollection<RoundedRectangle>` to a method that expects an `ICollection<Shape>` because that method may try to add other kinds of shapes.

> Sometimes, generics do not support covariance or contravariance even in situations where they would make sense. One reason for this is that although the CLR has supported variance since generics were introduced in .NET 2.0, C# did not fully support it until version 4.0. Before that release (in 2010) it was not possible to write a covariant or contravariant generic in C#, and you would have got an error if you had tried to apply the `in` and `out` keywords to type parameters in earlier versions. The .NET Framework class library was modified in 4.0: various classes that didn't previously support variance, but for which it made sense, were changed to offer it. However, there are plenty of

other class libraries out there, and if these were written before .NET 4.0, there's a good chance that they won't define any kind of variance.

Arrays are covariant, just like `IEnumerable<T>`. This is rather odd, given that we can write methods like the one in Example 6-18.

Example 6-18. Changing an element in an array

```
public static void UseBaseArray(Base[] bases)
{
    bases[0] = new Base();
}
```

If I were to call this with the code in Example 6-19, I would be making the same mistake as I did in Example 6-13, where I attempted to pass an `ICollection<Derived>` to a method that wanted to put something that was not `Derived` into the collection. But while Example 6-13 does not compile, Example 6-19 does, due to the surprising covariance of arrays.

Example 6-19. Passing an array with derived element type

```
Derived[] derivedBases = { new Derived(), new Derived() };
UseBaseArray(derivedBases);
```

This makes it look as though we could sneak a reference into an array to an object that is not an instance of the array's element type—in this case, putting a reference to a non-`Derived` `Base` in `Derived[]`. But that would be a violation of the type system. Does this mean the sky is falling?

In fact, C# correctly forbids such a violation, but it does so at runtime. Although a reference to an array of type `Derived[]` can be implicitly converted to a reference of type `Base[]`, any attempt to set an array element in a way that is inconsistent with the type system will throw an `ArrayTypeMismatchException`. So Example 6-18 would throw that exception when it tried to assign a reference to a `Base` into the `Derived[]` array.

Type safety is maintained, and rather conveniently, if we write a method that only reads from an array, we can pass arrays of some derived element type and it will work. The downside is that the CLR has to do extra work at runtime when you modify array elements to ensure that there is no type mismatch. It may be able to optimize the code to avoid having to check every single assignment, but there is still some overhead, meaning that arrays are not quite as efficient as they might be.

This somewhat peculiar arrangement dates back to the time before .NET had formalized concepts of covariance and contravariance—these came in with generics, which were introduced in .NET 2.0 (in 2005; it took C# half a decade to catch up with the framework). Perhaps if generics had been around from the start, arrays would be less odd, although having said that, their peculiar form of covariance is the only mechanism built into the framework that provides a way to pass a collection covariantly to a method that wants to read from it using indexing. There is no read-only indexed collection interface in the framework, and therefore no standard indexed collection interface with a covariant type parameter. (`IList<T>` is read/write.)

While we're on the subject of type compatibility and the implicit reference conversions that inheritance makes available, there's one more type we should look at: the `object` type.

System.Object

The `System.Object` type, or `object` as we usually call it in C#, is useful because it can act as a sort of universal container: a variable of this type can hold a reference to almost anything. I've mentioned this before, but I haven't yet explained why it's true. The reason this works is that almost everything derives from `object`.

If you do not specify a base class when writing a class, the C# compiler automatically uses `object` as the base. As we'll see shortly, it chooses different bases for certain kinds of types such as structs, but even those derive from `object` indirectly. (As ever, pointer types are an exception—these do not derive from `object`.)

The relationship between interfaces and objects is slightly more subtle. Interfaces do not derive from object, because an interface can only specify other interfaces as its bases. However, a reference of any interface type is implicitly convertible to a reference of type `object`. This conversion will always be valid, because all types that are capable of implementing interfaces ultimately derive from `object`. Moreover, C# chooses to make the members that the `object` class defines available through interface references even though the `object` class's members are not strictly speaking members of the interface. This means that any references of any kind always offer the following methods defined by `object`: `ToString`, `Equals`, `GetHashCode` and `GetType`.

The Ubiquitous Methods of object

I've used `ToString` in numerous examples already. The default implementation returns the object's type name, but many types provide their own implementation of `ToString`, returning a more useful textual representation of the object's current value. The numeric types return a decimal representation of their value for example, while `bool` returns either "True" or "False".

I discussed `Equals` and `GetHashCode` in Chapter 3, but I'll provide a quick recap here. `Equals` allows an object to be compared with any other object. The default implementation just performs an identity comparison, i.e., it returns `true` only when an object is compared with itself. Many types provide an `Equals` method that performs value-like comparison—for example, two distinct `string` objects may contain identical text, in which case they will report being equal to one another. (Should you need it, the identity-based comparison is always available through the `object` class's static `ReferenceEquals` method.) Incidentally, `object` also defines a static version of `Equals` that takes two arguments. This checks whether the arguments are `null`, returning `true` if both are `null`, `false` if only one or other is `null`, and otherwise it defers to the first argument's `Equals` method. And as discussed in Chapter 3, `GetHashCode` returns an integer that is a reduced representation of the object's value, which is used by hash-based mechanisms such as the `Dictionary<TKey, TValue>` collection class. Any pair of objects for which `Equals` returns `true` must return the same hash codes.

The `GetType` method provides a way to discover things about the object's type. It returns a reference of type `Type`. That's part of the reflection API, which is the subject of Chapter 13.

Besides these public members, available through any reference, `object` defines two more members that are not universally accessible. An object only has access to these members on itself. They are `Finalize` and `MemberwiseClone`. The `Finalize` method is called for you by the CLR to notify you that your object is no longer in use and the memory it uses is about to be reclaimed. In C# we do not normally work directly with the `Finalize` method, because C# presents this mechanism through destructors, as I'll show in Chapter 7. `MemberwiseClone` creates a new instance of the same type as your object, initialized with copies of all of your object's fields. If you need a way to create a clone of an object, this may be easier than writing code that copies all the contents across by hand.

The reason these last two methods are only available from inside the object is that you might not want other people cloning your object, and it would be unhelpful if external code could call the `Finalize` method, fooling your object into thinking that it was about to be freed if in fact it wasn't. The `object` class limits the accessibility of these members. But they're not private—that would mean that only the object class itself could access them, because private members are not visible even to derived classes. Instead, `object` makes theses members *protected*, an accessibility specifier designed for inheritance scenarios.

Accessibility and Inheritance

By now you will already be familiar with most of the accessibility levels available for types and their members. Elements marked as `public` are available to all, `private` members are only accessible from within the type that declared them, and `internal` members are available to any code defined in the same component.[1] But with inheritance, we get two other accessibility options.

A member marked as `protected` is available inside the type that defined it, and also inside of any derived types. But `protected` members are not visible from the outside—for code using an instance of your type, `protected` members may as well be `private`.

There's another protection level for type members: `protected internal`. (You can write `internal protected` if you prefer—the order makes no difference.) This makes the member more accessible than either `protected` or `internal` on its own. The member will be accessible to all derived types *and* to all code that shares an assembly.

> You may be wondering about the obvious conceptual counterpart: members that are only available to types that are both derived from *and* defined in the same component as the defining type. The CLR does support such a protection level, but C# does not provide any way to specify it.

You can specify `protected` or `protected internal` for any member of a type, not just methods. Even nested types can use these accessibility specifiers.

[1] More precisely, the same assembly. Chapter 12 describes assemblies.

Although `protected` (and `protected internal`) members are not available through an ordinary variable of the defining type, they are still part of the type's public API, in the sense that anyone who has access to your classes will be able to use these members. As with most languages that support a similar mechanism, `protected` members in C# are typically used to provide services that derived classes might find useful. If you write a `public` class that supports inheritance, then anyone can derive from it and gain access to its `protected` members. Removing or changing `protected` members would therefore risk breaking code that depends on your class just as surely as removing or changing `public` members would.

When you derive from a class, you cannot make your class more visible than its base. If you derive from an `internal` class, for example, you cannot declare your class to be `public`. Your base class forms part of your class's API, and so anyone wishing to use your class will also in effect be using its base class, so if the base is inaccessible, your class will also be inaccessible, which is why C# does not permit a class to be more visible than its base. For example, if you derive from a `protected` nested class, your derived class could be `protected` or `private`, but not `public`, nor `internal`, nor `protected internal`.

> This restriction does not apply to the interfaces you implement. A `public` class is free to implement `internal` or `private` interfaces. However, it does apply to an interface's bases: a `public` interface cannot derive from an `internal` interface.

When defining methods, there's another keyword you can add for the benefit of derived types: `virtual`.

Virtual Methods

A *virtual method* is one that a derived type can replace. Several of the methods defined by `object` are virtual: the `ToString`, `Equals`, `GetHashCode`, and `Finalize` methods are all designed to be replaced. The code required to produce a useful textual representation of an object's value will differ considerably from one type to another, as will the logic required to determine equality and produce a hash code. Types typically only define a finalizer if they need to do some specialized cleanup work when they go out of use.

Not all methods are virtual. In fact, C# makes methods non-virtual by default. The `object` class's `GetType` method is not virtual, so you can always trust the information it returns to you because you know that you're calling the `GetType` method supplied by the .NET Framework, and not some type-specific substitute designed to fool you. To declare that a method should be virtual, use the `virtual` keyword as Example 6-20 shows.

Example 6-20. A class with a virtual method

```
public class BaseWithVirtual
{
    public virtual void ShowMessage()
    {
        Console.WriteLine("Hello from BaseWithVirtual");
    }
```

```
}
```

There's nothing unusual about the syntax for invoking a virtual method. As Example 6-21 shows, it looks just like calling any other method.

Example 6-21. Using a virtual method

```
public static void CallVirtualMethod(BaseWithVirtual o)
{
    o.ShowMessage();
}
```

The difference between virtual and non-virtual method invocations is that a virtual method call decides at runtime which method to invoke. The code in Example 6-21 will, in effect, inspect the object passed in, and if the object's type supplies its own implementation of `ShowMessage`, it will call that instead of the one defined in `BaseWithVirtual`. The method is chosen based on the actual type the target object turns out to have at runtime, and not the static type (determined at compile time) of the expression that refers to the target object.

> Since virtual method invocation selects the method based on the type of the object on which you invoke the method, static methods cannot be virtual.

Derived types are not obliged to replace virtual methods of course. Example 6-22 shows two classes that derive from the one in Example 6-20. The first leaves the base class's implementation of `ShowMessage` in place. The second overrides it. Note the `override` keyword—C# requires us to state explicitly that we are intending to override a virtual method.

Example 6-22. Overriding virtual methods

```
public class DeriveWithoutOverride : BaseWithVirtual
{
}

public class DeriveAndOverride : BaseWithVirtual
{
    public override void ShowMessage()
    {
        Console.WriteLine("This is an override");
    }
}
```

Having defined these types, we can now use the method in Example 6-21. Example 6-23 calls it three times, passing in a different type of object each time.

Example 6-23. Exploiting virtual methods

```
CallVirtualMethod(new BaseWithVirtual());
CallVirtualMethod(new DeriveWithoutOverride());
CallVirtualMethod(new DeriveAndOverride());
```

This produces the following output:

```
Hello from BaseWithVirtual
Hello from BaseWithVirtual
This is an override
```

Obviously, when we pass an instance of the base class, we get the output that the base class's `ShowMessage` method prints. We also get that with the derived class that has not supplied an override. It's only the final class, which overrides the method, that produces different output.

This is all very similar to interfaces—virtual methods provide another way to write polymorphic code. Example 6-21 can use a variety of different types, which can modify the behavior if necessary. The big difference is that the base class can supply a default implementation for each virtual method, something that interfaces cannot do.

Abstract Methods

You can define a virtual method without providing a default implementation. C# calls this an *abstract method*. If a class contains one or more abstract methods, the class is incomplete because it doesn't provide all of the methods it defines. Classes of this kind are also described as being abstract, and it is not possible to construct instances of an abstract class—attempting to use the `new` operator with an abstract class will cause a compiler error. Sometimes when discussing classes it's useful to make it clear that some particular class is not abstract, for which we normally use the term *concrete class*.

If you derive from an abstract class, then unless you provide implementations for all the abstract methods, your derived class will also be abstract. You must state your intention to write an abstract class with the `abstract` keyword—if this is absent from a class that has unimplemented abstract methods (either ones it has defined itself, or ones it has inherited from its base class) the C# compiler will report an error. Example 6-24 shows an abstract class that defines a single abstract method. Abstract methods are by definition virtual; there wouldn't be much use in defining a method that has no body, and which didn't provide a way for derived classes to supply a body.

Example 6-24. An abstract class

```
public abstract class AbstractBase
{
    public abstract void ShowMessage();
}
```

As with interface members, abstract method declarations just define the signature, and do not contain a body. Unlike with interfaces, each abstract member has its own accessibility—you can declare abstract methods as `public`, `internal`, `protected internal`, or `protected`. (It makes no sense to make an abstract or virtual method `private`, because the method will be inaccessible to derived types and therefore impossible to override.)

> Although classes that contain abstract methods are required to be abstract, the converse is not true. It is legal, albeit unusual, to define a class as abstract even if it would be a viable non-abstract class. This prevents the class from being constructed. A class that derives from this will be concrete without needing to override any abstract methods.

Abstract classes have the option to declare that they implement an interface without needing to provide a full implementation. You can't just declare the interface and omit members though. You must explicitly declare all of its members, marking any that you want to leave unimplemented as being abstract, as Example 6-25 shows. This forces derived types to supply the implementation.

Example 6-25. Abstract interface implementation

```
public abstract class MustBeComparable : IComparable<string>
{
    public abstract int CompareTo(string other);
}
```

There's clearly some overlap between abstract classes and interfaces. Both provide a way to define an abstract type that code can use without needing to know the exact type that will be supplied at runtime. Each option has its pros and cons. Interfaces have the advantage that a single type can implement multiple interfaces; a class only gets to specify a single base class. But abstract classes can provide default implementations for some or even all methods. This makes abstract classes more amenable to evolution as you release new versions of your code.

Imagine what would happen if you had written and released a library that defined some public interfaces, and in the second release of the library, you decided that you wanted to add some new members to some of these interfaces. This might not cause a problem for customers using your code—any place where they use a reference of that interface type will be unaffected by the addition of new features. However, what if some of your customers have written implementations of your interfaces? Suppose, for example, that in a future version of .NET, Microsoft decided to add a new member to the `IEnumerable<T>` interface.

That would be a disaster. This interface is widely used, but also widely implemented. Classes that already implement `IEnumerable<T>` would become invalid because they would not provide this new member, so old code would fail to compile, and code already compiled would throw `MissingMethodException` errors at runtime. Or worse, some classes might by chance already have a member with the same name and signature as the newly-added method. The compiler would treat that existing member as part of the implementation of the interface, even though the developer who wrote the method did not write it with that intention. So unless the existing code coincidentally happens to do exactly what the new member requires, we'd have a problem, and we wouldn't get a compiler error.

Consequently, the widely accepted rule is that you do not alter interfaces once they have been published. If you have complete control over all of the code that uses an interface, you can get away with modifying the interface because you can make any necessary modifications to code that consumes it. But once the interface has become available for use in codebases you do not control—once it has been published—it's no longer possible to change it without being likely to break someone else's code.

Abstract base classes do not have to suffer from this problem. Obviously, introducing new abstract members would cause exactly the same sorts of issues, but introducing new virtual methods is considerably less problematic. With a non-abstract virtual method, you supply a default implementation, so it doesn't matter if a derived class does not implement it.

But what if, after releasing version 1.0 of a component, you add a new virtual method in v1.1 which turns out to have the same name and signature as a method that one of your customers happens to have added in a derived class? Perhaps in version 1.0, your component defines the rather uninteresting base class shown in Example 6-26.

Example 6-26. Base type version 1.0

```
public class LibraryBase
```

```
{
}
```

If you release this library, perhaps as a product in its own right, or maybe as part of some SDK for your application, a customer might write a derived type such as the one in Example 6-27. They've written a `Start` method which is clearly not meant to override anything in the base class.

Example 6-27. Class derived from version 1.0 base

```
public class CustomerDerived : LibraryBase
{
    public void Start()
    {
        Console.WriteLine("Derived type's Start method");
    }
}
```

Of course, you won't necessarily get to see every line of code that your customers write, so you might be unaware of that `Start` method. So in version 1.1 of your component, you might decide to add a new virtual method, also called `Start`, as Example 6-28 shows.

Example 6-28. Base type version 1.1

```
public class LibraryBase
{
    public virtual void Start() { }
}
```

Imagine that your system calls this method as part of some initialization procedure. You've defined a default empty implementation so that types derived from `LibraryBase` that don't need to take part in that procedure don't have to do anything. Types that wish to participate will override this method. But what happens with the class in Example 6-27? Clearly the developer who wrote that did not intend to participate in your new initialization mechanism, because that didn't even exist at the time at which the code was written. It could be bad if your code calls the `CustomerDerived` class's `Start` method, because the developer presumably only expects it to be called when her code decides to call it. Fortunately, the compiler will detect this problem. If the customer attempts to compile Example 6-27 against version 1.1 of your library (Example 6-28) the compiler will warn her that something is not right:

```
warning CS0114: 'CustomerDerived.Start()' hides inherited member
'LibraryBase.Start()'. To make the current member override that implementation,
add the override keyword. Otherwise add the new keyword.
```

This is why the C# compiler requires the `override` keyword when we replace virtual methods. It wants to know whether we were intending to override an existing method, so that if we weren't, it can warn us about collisions.

It reports a warning rather than an error, because it provides a behavior that is likely to be safe when this situation has arisen due to the release of a new version of a library. The compiler guesses, correctly in this case, that the developer who wrote the `CustomerDerived` type didn't mean to override the `LibraryBase` class's `Start` method. So rather than having the `CustomerDerived` type's `Start` method override the base class's virtual method, it *hides* it. A derived type is said to hide a member of a base class when it introduces a new member with the same name.

Hiding methods is quite different than overriding them. When hiding occurs, the base method is not replaced. Example 6-29 shows how the hidden `Start` method remains available. It creates a `CustomerDerived` object, and places a reference to that object in two variables of different types: one of type `CustomerDerived`, and one of type `LibraryBase`. It then calls `Start` through each of these.

Example 6-29. Hidden vs virtual method

```
var d = new CustomerDerived();
LibraryBase b = d;

d.Start();
b.Start();
```

When using the `d` variable, the call to `Start` ends up calling the derived type's `Start` method, the one that has hidden the base member. But the `b` variable's type is `LibraryBase`, so that invokes the base `Start` method. If `CustomerDerived` had overridden the base class's `Start` method instead of hiding it, both of those method calls would have invoked the override.

When name collisions occur because of a new library version, this hiding behavior is usually the right thing to do. If the customer's code has a variable of type `CustomerDerived`, then that code will want to invoke the `Start` method specific to that derived type. However, the compiler produces a warning, because it doesn't know for certain that this is the reason for the problem. It might be that you did mean to override the method, and you just forgot to write the `override` keyword.

Like many developers, I don't like to see compiler warnings, and I try to avoid checking in code that produces them. But what should you do if a new library version puts you in this situation? The best long-term solution is probably to change the name of the method in your derived class so that it doesn't clash with the method in the new version of the library. However, if you're up against a deadline, you may want a more expedient solution. So C# lets you declare that you know that there's a name clash, and that you definitely want to hide the base member, not override it. As Example 6-30 shows, you can use the `new` keyword to state that you're aware of the issue, and you definitely want to hide the base class member. The code will still behave in the same way, but you'll no longer get the warning because you've assured the compiler that you know what's going on. But this is an issue you should fix at some point, because sooner or later, the existence of two methods with the same name on the same type that mean different things is likely to cause confusion.

Example 6-30. Avoiding warnings when hiding members

```
public class CustomerDerived : LibraryBase
{
    public new void Start()
    {
        Console.WriteLine("Derived type's Start method");
    }
}
```

Just occasionally, you may see the `new` keyword used in this way for reasons other than handling library versioning issues. For example, the `ISet<T>` interface that I showed in Chapter 5 uses it to introduce a new `Add` method. `ISet<T>` derives from `ICollection<T>`, an interface that already provides an `Add` method, which takes an

instance of `T`, and has a `void` return type. `ISet<T>` makes a subtle change to this shown in Example 6-31.

Example 6-31. Hiding to change the signature

```
public interface ISet<T> : ICollection<T>
{
    new bool Add(T item);
    ... other members omitted for clarity
}
```

The `ISet<T>` interface's `Add` method tells you whether the item you just added was already in the set, something the base `ICollection<T>` interface's `Add` method doesn't support. `ISet<T>` needs its `Add` to have a different return type, `bool` instead of `void`, so it defines `Add` with the `new` keyword to indicate that it should hide the `ICollection<T>` one. Both methods are still available—if you have two variables, one of type `ICollection<T>` and the other of type `ISet<T>`, both referring to the same object, you'll be able to access the `void Add` through the former, and the `bool Add` through the latter. (Microsoft didn't have to do this. They could have called the new `Add` method something else—`AddIfNotPresent`, for example. But it's arguably less confusing just to have the one method name for adding things to a collection, particularly since you're free to ignore the return value, at which point the new `Add` looks indistinguishable from the old one. And most `ISet<T>` implementations will implement the `ICollection<T>.Add` method by calling straight through to the `ISet<T>.Add` method, so it makes sense that they have the same name.)

So far, I've only discussed method hiding in the context of compiling old code against a new version of a library. What happens if you have old code compiled against an old library but which ends up running against a new version? That's a scenario you are highly likely to run into when the library in question is the .NET Framework class library. Suppose you are using 3rd party components that you only have in binary form (e.g., ones you've bought from a company that does not supply source code). The supplier will have built these to use some particular version of .NET. If you upgrade your application to run with a new version of .NET, you might not be able to get hold of newer versions of the 3rd party components—maybe the vendor hasn't released them yet, or perhaps they've gone out of business.

If the components you're using were compiled for, say, .NET 4.0, and you use them in a project built for .NET 4.5, all of those older components will end up using the .NET 4.5 versions of the framework class library. The .NET Framework has a versioning policy that arranges for all the components that a particular program uses to get the same version of the framework class library, regardless of which version any individual component may have been built for. So it's entirely possible that some component, *OldControls.dll*, contains classes that derive from classes in the .NET 4.0 Framework, and which define members that collide with the names of members newly added in .NET 4.5.

This is more or less the same scenario as I described earlier, except that the code that was written for an older version of a library is not going to be recompiled. We're not going to get a compiler warning about hiding a method, because that would involve running the compiler, and we only have the binary for the relevant component. What happens now?

Fortunately, we don't need the old component to be recompiled. The C# compiler sets various flags in the compiled output for each method it compiles, indicating things like whether the method is virtual or not, and whether the method was intended to override

some method in the base class. When you put the `new` keyword on a method, the compiler sets a flag indicating that the method is not meant to override anything in the base class. The CLR calls this the *newslot* flag. When C# compiles a method such as the one in Example 6-27, which does not specify either `override` or `new`, it also sets this same *newslot* flag for that method, because at the time the method was compiled, there was no method of the same name on the base class. As far as both the developer and the compiler were concerned, the `CustomerDerived` class's `Start` was written as a brand new method which was not connected to anything on the base class.

So when this old component gets loaded in conjunction with a new version of the framework library defining the base class, the CLR can see what was intended—it can see that as far as the author of the `CustomerDerived` class was concerned `Start` is not meant to override anything. It therefore treats `CustomerDerived.Start` as a distinct method from `LibraryBase.Start`—it hides the base method just like it did when we were able to recompile.

By the way, everything I've said about virtual methods can also apply to properties, because a property's accessors are just methods. So you can define virtual properties, and derived classes can override or hide these in exactly the same way as with methods. I won't be getting to properties until Chapter 9, but those are also methods in disguise, so those can also be virtual.

Just occasionally, you may want to write a class that overrides a virtual method, and then prevents derived classes from overriding it again. For this, C# defines the `sealed` keyword, and in fact, it's not just methods that can be sealed.

Sealed Methods and Classes

Virtual methods are deliberately open to modification through inheritance. A sealed method is the opposite—it is one that cannot be overridden. Methods are sealed by default in C#: methods cannot be overridden unless declared virtual. But when you override a method, you can seal it, closing it off for further modification. Example 6-32 uses this to provide a custom `ToString` implementation that cannot be further overridden by derived classes.

Example 6-32. A sealed method

```
public class FixedToString
{
    public sealed override string ToString()
    {
        return "Arf arf!";
    }
}
```

You can also seal an entire class, preventing anyone from deriving from it. Example 6-33 shows a class that not only does nothing, it prevents anyone from extending it to do something useful. (You'd normally only seal a class that does something. This example is just to illustrate where the keyword goes.)

Example 6-33. A sealed class

```
public sealed class EndOfTheLine
{
}
```

Some types are inherently sealed. Value types, for example, do not support inheritance, so structs and enums are effectively sealed. The built-in `string` class is also sealed.

There are two normal reasons for sealing either classes or methods. One is that you want to guarantee some particular invariant, and if you leave your type open to modification, you will not be able to guarantee that invariant. For example, instances of the `string` type are immutable. The `string` type itself does not provide any way to modify an instance's value, and because nobody can derive from `string`, you can guarantee that if you have a reference of type `string`, you have a reference to an immutable object. This makes it safe to use in scenarios where you don't want the value to change—for example, when you use an object as a key to a dictionary (or anything else that relies on a hash code) you need the value not to change because if the hash code changes while the item is in use as a key, the container will malfunction.

The other usual reason for leaving things sealed is that designing types that can successfully be modified through inheritance is hard, particularly if your type will be used outside of your own organization. Simply opening things up for modification is not sufficient—if you decide to make all your methods virtual, it might make it easy for people using your type to modify its behavior, but you will have made a rod for your back when it comes to maintaining the base class. Unless you control all of the code that derives from your class, it will be almost impossible to change anything in the base because you will never know which methods may have been overridden in derived classes, making it very hard to ensure that your class's internal state is consistent at all times. Developers writing derived types will doubtless do their best not to break things, but they will inevitably rely on aspects of your class's behavior that are undocumented. So in opening up every aspect of your class for modification through inheritance, you rob yourself of the freedom to change your class.

You should typically be very selective about which methods, if any, you make virtual. And you should also document whether callers are allowed to replace the method completely, or whether they are required to call the base implementation as part of their override. Speaking of which, how do you do that?

Accessing Base Members

Everything that is in scope in a base class and which is not private will also be in scope and accessible in a derived type. So for the most part, if you want to access some member of the base class, you just access it as if it were a normal member of your class. You can either access members through the `this` reference, or just refer to them by name without qualification.

However, there are some situations in which it is useful to be able to refer explicitly to a base class member. In particular, if you have overridden a method, calling that method by name will invoke your override. If you want to call back to the original method that you overrode, there's a special syntax for that, shown in Example 6-34.

Example 6-34. Calling the base method after overriding

```
public class CustomerDerived : LibraryBase
{
    public override void Start()
    {
        Console.WriteLine("Derived type's Start method");
```

```
        base.Start();
    }
}
```

By using the `base` keyword we are opting out of the normal virtual method dispatch mechanism. If we had written just `Start()`, that would have been a recursive call, which would be undesirable here. By writing `base.Start()`, we get the method that would have been available on an instance of the base class, the method that we overrode.

In this example I've called the base class's implementation after completing my own work. C# doesn't care when you call the base—you could call it as the first thing the method does, or as the last, or half way through the method. You could even call it several times, or not at all. It is up to the author of the base class to document whether and when the base class implementation of the method should be called by an override.

You can use the `base` keyword for other members too, such as properties and events. However, access to base constructors works slightly differently.

Inheritance and Construction

Although a derived class inherits all the members of its base class, this does not mean the same thing for constructors as it does for everything else. With other members, if they are public in the base class, they will be public members of the derived class too, accessible to anyone who uses your derived class. But constructors are special, because someone using your class cannot construct it by using one of the constructors defined by the base class.

It's obvious enough why that should be: if you want an instance of some type `D`, then you'll want it to be a fully-fledged `D` with everything in it properly initialized. Suppose that `D` derives from `B`. If you were able to use one of `B`'s constructors directly, it wouldn't do anything to the parts specific to `D`. A base class's constructor won't know about any of the fields defined by a derived class, so it cannot initialize them. If you want a `D`, you'll need a constructor that knows how to initialize a `D`. So with a derived class, you can only use the constructors offered by that derived class, regardless of what constructors the base class might provide.

In the examples I've shown so far in this chapter, I've been able to ignore this because of the default constructor that C# provides. As you saw in Chapter 3, if you don't write a constructor, C# writes one for you that takes no arguments. It does this for derived classes too, and the generated constructor will invoke the default constructor of the base class. But this changes if I start writing my own constructors. Example 6-35 defines a pair of classes, where the base defines an explicit no-arguments constructor, and the derived class defines one that requires an argument.

Example 6-35. No default constructor in derived class

```
public class BaseWithZeroArgCtor
{
    public BaseWithZeroArgCtor()
    {
        Console.WriteLine("Base constructor");
    }
}
```

```
public class DerivedNoDefaultCtor : BaseWithZeroArgCtor
{
    public DerivedNoDefaultCtor(int i)
    {
        Console.WriteLine("Derived constructor");
    }
}
```

Because the base class has a zero-argument constructor, I can construct it with `new BaseWithZeroArgCtor()`. But I cannot do this with the derived type—I can only construct that by passing an argument, e.g. `new DerivedNoDefaultCtor(123)`. So as far as the publically visible API of `DerivedNoDefaultCtor` is concerned, the derived class appears not to have inherited its base class's constructor.

However, it has in fact inherited it, as you can see by looking at the output you get if you construct an instance of the derived type:

```
Base constructor
Derived constructor
```

When I construct an instance of `DerivedNoDefaultCtor`, the base class's constructor runs immediately before the derived class's constructor. Since the base constructor ran, clearly it was available. All of the base class's constructors are available to a derived type, but they must be accessed via derived constructors. In Example 6-35, the base constructor was invoked implicitly: all constructors are required to invoke a constructor on their base class, and if you don't specify which to invoke, the compiler invokes the base's zero-argument constructor for you.

What if the base doesn't define a parameterless constructor? In that case you'll get a compiler error if you derive a class that does not specify which constructor to call. Example 6-36 shows a base class with no zero-argument constructor. (The presence of any explicit constructors disables the compiler's normal generation of a default constructor, and since this base class only supplies a constructor that takes arguments, this means there is no zero-argument constructor.) It also shows and a derived class with two constructors, both of which call into the base constructor explicitly, using the `base` keyword.

Example 6-36. Invoking a base constructor explicitly

```
public class BaseNoDefaultCtor
{
    public BaseNoDefaultCtor(int i)
    {
        Console.WriteLine("Base constructor: " + i);
    }
}

public class DerivedCallingBaseCtor : BaseNoDefaultCtor
{
    public DerivedCallingBaseCtor()
        : base(123)
    {
        Console.WriteLine("Derived constructor (default)");
    }

    public DerivedCallingBaseCtor(int i)
        : base(i)
```

```
    {
        Console.WriteLine("Derived constructor: " + i);
    }
}
```

The derived class here decides to supply a parameterless constructor even though the base class doesn't have one—it supplies a fixed value for the argument the base requires. The second just passes its argument through to the base.

> Here's a frequently asked question: how do I provide all the same constructors as my base class, just passing all the arguments straight through? The answer is: write all the constructors by hand. There is no way to get C# to generate a set of constructors in a derived class that look identical to the ones that the base class offers. You need to do it the long-winded way.

As Chapter 3 showed, a class's field initializers run before its constructor. The picture is slightly more complicated once inheritance is involved because there are multiple classes and multiple constructors. The easiest way to predict what will happen is to understand that although instance field initializers and constructors have separate syntax, C# ends up compiling all the initialization code for a particular class into the constructor. This code performs the following steps: first, it runs any field initializers specific to this class (so this step does not include base field initializers—the base class will take care of itself); next, it calls the base class constructor; finally, it runs the body of the constructor. The upshot of this is that in a derived class, your instance field initializers will run before any base class construction has occurred—not just before the base constructor body, but even before the base's instance fields have been initialized. Example 6-37 illustrates this.

Example 6-37. Exploring construction order

```
public class BaseInit
{
    protected static int Init(string message)
    {
        Console.WriteLine(message);
        return 1;
    }

    private int b1 = Init("Base field b1");

    public BaseInit()
    {
        Init("Base constructor");
    }

    private int b2 = Init("Base field b2");
}

public class DerivedInit : BaseInit
{
    private int d1 = Init("Derived field d1");

    public DerivedInit()
    {
        Init("Derived constructor");
    }
```

```
    private int d2 = Init("Derived field d2");
}
```

I've put the field initializers on either side of the constructor just to prove that their position relative to non-field members is irrelevant. The order of the fields matters, but only with respect to one another. Constructing an instance of the `DerivedInit` class produces this output:

```
Derived field d1
Derived field d2
Base field b1
Base field b2
Base constructor
Derived constructor
```

This verifies that the derived type's field initializers run first, and then the base field initializers, followed by the base constructor and then finally the derived constructor. In other words, although constructor bodies start with the base class, instance field initialization happens in reverse.

That's why you don't get to invoke instance methods in field initializers. Static methods are available, but instance methods are not, because the class is a long way from being ready. It could be problematic if one of the derived type's field initializers were able to invoke a method on the base class, because the base class has performed no initialization at all at that point—not only has its constructor body not run, its field initializers haven't run either. If instance methods were available during this phase, we'd have to write all of our code to be very defensive, because we could not assume that our fields contain anything useful.

As you can see, the constructor bodies run relatively late in the process, which is why we are allowed to invoke methods from them. But there's still potential danger here. What if the base class defines a virtual method and invokes that method on itself in its constructor? If the derived type overrides that, we'll be invoking the method before the derived type's constructor body has run. (Its field initializers will have run at that point, though. In fact, this is the main benefit of the fact the field initializers run in what seems to be reverse order—it means that derived classes have a way of performing some initialization before the base class's constructor has a chance to invoke a virtual method.) If you're familiar with C++ you might hazard a guess that when the base constructor invokes a virtual method, it'll run the base implementation. But C# does it differently: a base class's constructor will invoke the derived class's override in that case. This is not necessarily a problem, and it can occasionally be useful, but it means you need to think carefully and document your assumptions clearly if you want your object to invoke virtual methods on itself during construction.

Special Base Types

The .NET Framework class library defines a few base types that have special significance in C#. The most obvious is `System.Object`, which I've already described in some detail.

There's also `System.ValueType`. This is the abstract base type of all value types, so any `struct` you define, and also all of the built-in value types such as `int` and `bool`

derive from `ValueType`. Ironically, `ValueType` itself is a reference type; only types that derive from `ValueType` are value types. Like most types, `ValueType` derives from `System.Object`. There is an obvious conceptual difficulty here: in general, derived classes are everything their base class is, plus whatever functionality they add. So given that `object` and `ValueType` are both reference types, it may seem odd that types derived from `ValueType` are not. And for that matter it's not obvious how an `object` variable can hold a reference to an instance of something that's not a reference type. I will resolve all of these issues in Chapter 7.

C# does not permit you to derive explicitly from `ValueType`. If you want to write a type that derives from `ValueType`, that's what the `struct` keyword is for. You can declare a variable of type `ValueType`, but since the type doesn't define any public members, a `ValueType` reference doesn't enable anything you can't do with an `object` reference. The only observable difference is that with a variable of that type you can assign instances of any value type into it but not instances of a reference type. Aside from that, it's identical to `object`. Consequently, it's fairly rare to see `ValueType` mentioned explicitly in C# code.

Enumeration types also all derive from a common abstract base type: `System.Enum`. Since enums are value types, you'll be unsurprised to hear that `Enum` derives from `ValueType`. As with `ValueType` you would never derive from `Enum` explicitly—you use the `enum` keyword for that. Unlike `ValueType`, `Enum` does add some useful members. For example, its `GetValues` method returns an array of all the enumeration's values, while `GetNames` returns an array with all those values converted to strings. It also offers some static methods, such as `Parse`, which converts from the string representation back to the enumeration value.

As Chapter 5 described, arrays all derive from a common base class, `System.Array`, and you've already seen the features that offers.

The `System.Exception` base class is special: C# only permits you to throw exceptions of this type or types that derive from this. (Exceptions are the topic of Chapter 8.)

Delegate types all derive from a common base type `System.MulticastDelegate`, which in turn derives from `System.Delegate`. I'll discuss these in Chapter 9.

Those are all the base types that correspond to special types in the CTS. There's one more base type to which the C# compiler assigns special significance, and that's `System.Attribute`. In Chapter 1, I applied certain annotations to methods and classes to tell the unit test framework to treat them specially. These attributes all correspond to types—so I applied the `[TestClass]` attribute to a class, and in doing so I was using a type called `TestClassAttribute`. Types designed to be used as attributes are all required to derive from `System.Attribute`. Some of them are recognized by the compiler—for example, there are some that control the version numbers that the compiler puts into the file headers of the EXE and DLL files it produces. I'll show all of this in Chapter 15.

Summary

C# supports single implementation inheritance, and only with classes—you cannot derive from a struct at all. However, interfaces can declare multiple bases, and a class can implement multiple interfaces. Implicit reference conversions exist from derived types to base types, and generic types can choose to offer related implicit reference conversions using either covariance or contravariance. All types derive from `System.Object`, guaranteeing that certain standard members are available on all variables. We saw how virtual methods allow derived classes to modify selected members of their bases, and how sealing can disable that. We also looked at the relationship between a derived type and its base when it comes to accessing members, and constructors in particular.

Our exploration of inheritance is complete, but it has raised some new issues such as the relationship between value types and references, and the role of finalizers. So in the next chapter, I'll talk about the relationship between references and an object's lifecycle, along with the way the CLR bridges the gap between references and value types.

7

Object Lifetime

One of the benefits of .NET's managed execution model is that the runtime can automate most of your application's memory management. I have shown numerous examples that create objects with the `new` keyword, and none of these have explicitly freed the memory consumed by these objects.

In most cases, you do not need to take any action to reclaim memory. The runtime provides a garbage collector (GC), a mechanism that automatically discovers when objects are no longer in use, and recovers the memory they had been occupying so that it can be used for new objects. However, there are certain usage patterns that can cause performance issues, or even defeat the GC entirely, so it's useful to understand how it works. This is particularly important with long-running processes that could run for days. (Short-lived processes may be able to tolerate a few memory leaks.)

Although most code can remain oblivious to the garbage collector, it is sometimes useful to be notified when an object is about to be collected, which C# makes possible through destructors. The underlying runtime mechanism that supports this is called finalization, and it has some important pitfalls, so I'll show how and how not to use destructors.

The garbage collector is designed to manage memory efficiently, but memory is not the only limited resource you may need to deal with. Some things have a small memory footprint in the CLR, but represent something relatively expensive, such as a database connection or a handle from a Win32-style API. The GC doesn't always deal with these effectively, so I'll explain `IDisposable`, the interface designed for dealing with things that need to be freed more urgently than memory.

Value types often have completely different rules governing their lifetime—some local variable values live only for as long as their containing method runs, for example. Nonetheless, value types sometimes end up acting like reference types, and being managed by the garbage collector. I will discuss why that can be useful, and I will explain the *boxing* mechanism that makes it possible.

Garbage Collection

The CLR maintains a *heap*, a service that provides memory for objects and values whose lifetime is managed by the garbage collector. Each time you construct an instance of a class with `new`, the CLR allocates a new heap block for that object. The garbage collector decides when to deallocate that block.

A heap block contains all the non-static fields for the object. The CLR also adds a header which is not directly visible to your program. This includes a pointer to a structure describing the object's type. This supports operations that depend on the real type of an object. For example, if you call `GetType` on a reference of type `object`, the CLR uses this pointer to find out the type. It's also used to work out which method to use when you invoke a virtual method or an interface member. The CLR also uses this to know how large the heap block is—the header does not include the block size, because the CLR can work that out from the object's type. (Most types are fixed size. There are only two exceptions, strings and arrays, which the CLR handles as special cases.) The header contains one other field which is used for a variety of diverse purposes, including multithreaded synchronization, and default hash code generation. Heap block headers are just an implementation detail, and other CLI implementations could choose different strategies. However, it's useful to know what the overhead is. On a 32-bit system, the header is 8 bytes long, and if you're running in a 64-bit process, it takes 16 bytes. So an object that contained just one field of type `double` (an 8-byte type) would consume 16 bytes in a 32-bit process, and 24 bytes in a 64-bit process.

Although objects (i.e., instances of a class) always live on the heap, instances of value types are different: some live on the heap, and some don't. The CLR stores some value-typed local variables on the stack, for example, but if the value is in an instance field of a class, the class instance will live on the heap, and that value will therefore live inside that object on the heap. And in some cases, a value will have an entire heap block to itself.

If you are accessing something through a reference-typed variable, then you are accessing something on the heap. This does not include all `out` or `ref` style method arguments by the way. Although those are references of a kind, a `ref int` argument is a reference to a value type, and that's not the same thing as a reference type. For the purposes of this discussion, a reference is something you can store in a variable of a type that derives from `object`, but which does not derive from `ValueType`.

The managed execution model used by C# (and all .NET languages) means the CLR knows about every heap block your code creates, and also about every field, variable, and array element in which your program stores references. This information enables the runtime to determine at any time which objects are *reachable*, i.e., those which the program could conceivably get access to in order to use its fields and other members. If an object is not reachable, then by definition the program will never be able to use it again. To illustrate how the CLR determines reachability, I've written a simple method that fetches web pages from my blog, shown in Example 7-1.

Example 7-1. Using and discarding objects

```
public static string GetBlogEntry(string relativeUri)
{
    var baseUri = new Uri("http://www.interact-sw.co.uk/iangblog/");
    var fullUri = new Uri(baseUri, relativeUri);
    using (var w = new WebClient())
    {
```

```
        return w.DownloadString(fullUri);
    }
}
```

The CLR analyses the way in which we use local variables and method arguments. Although the `relativeUri` argument is in scope for the whole method, we use it just once as an argument when constructing the second `Uri`, and then never use it again. A variable is described as *live* from the first point where it receives a value up until the last point at which it is used. Method arguments are live from the start of the method until their final usage, unless they are unused, in which case they are never live. Variables become live later—`baseUri` becomes live once it has been assigned its initial value, and then ceases to be live with its final usage, at the same point as `relativeUri`. Liveness is an important property in determining whether a particular object is still in use.

To see the role that liveness plays, suppose that when Example 7-1 reaches the line that constructs the `WebClient`, the CLR doesn't have enough free memory to hold the new object. It could request more memory from the OS at this point, but it also has the option to try and free up memory from objects that are no longer in use, meaning that our program wouldn't need to consume any more memory than it's already using.[1] The next section describes the process that the CLR uses when it takes that second option.

Determining Reachability

The CLR starts by determining all of the *root references* in your program. A root is a storage location, such as a local variable, which could contain a reference, and which is known to have been initialized, and that your program could use at some point in the future, without needing to go via some other object reference first. Not all storage locations are considered to be roots. If an object contains an instance field of some reference type, that field is not a root because before you can use it, you'd need to get hold of a reference to the containing object, and it's possible that the object itself is not reachable. However, a reference type static field is a root reference, because the program can read the value in that field at any time—the only situation in which that field will become inaccessible in the future is when the program exits.

Local variables are more interesting. (So are method arguments; everything I say about locals in this section also applies to arguments.) Sometimes they are roots, but sometimes not. It depends on where exactly in the method execution has got to. A local variable can only be a root if the flow of execution is currently inside the region in which that variable is live. So in Example 7-1, `baseUri` is only a root reference after it has had its initial value assigned, and before the call to construct the second `Uri`, which is a rather narrow window. The `fullUri` variable is a root reference for slightly longer because it becomes live after receiving its initial value, and continues to be live during the construction of the `WebClient` on the following line, and its liveness only ends once `DownloadString` has been called.

[1] The CLR doesn't always wait until runs out of memory. I will discuss the details later. For now, the important point is that from time to time, it will try to free up some space.

> When a variable's last use is as an argument in a method or constructor invocation, it ceases to be live when the method call begins. At that point, the method being called takes over—its own arguments are live at the start. However, they will typically cease to be live before the method returns. This means that in Example 7-1, the object referred to by `fullUri` may cease to be accessible through any root references before the call to `DownloadString` returns.

Since the set of live variables changes as the program executes, the set of root references also evolves, so the CLR needs to be able to form a snapshot of the relevant program state. The exact details are undocumented, but the garbage collector is able to suspend all threads that are running managed code when necessary to guarantee correct behavior.

Live variables and static fields are not the only kinds of roots. Temporary objects created as a result of evaluating expressions need to stay alive for as long as necessary to complete the evaluation, so there can be some root references that don't correspond directly to any named entities in your code. And there are other types of root. For example, the `GCHandle` class lets you create new roots explicitly, which can be useful in interop scenarios, to enable some unmanaged code to get access to a particular object. There are also situations in which roots are created implicitly. Interop with COM objects can establish root references without explicit use of `GCHandle`—if the CLR needs to generate a COM wrapper for one of your .NET objects, that wrapper will effectively be a root reference. Calls into unmanaged code may also involve passing pointers to memory on the heap, which will mean that the relevant heap block needs to be treated as reachable for the duration of the call. The CLI specification does not dictate the exact list of ways in which root references come into existence, and the CLR does not comprehensively document all the kinds it can create, but the broad principle is that roots will exist where necessary to ensure that objects that are still in use remain reachable.

Having built up a complete list of current root references for all threads, the garbage collector works out which objects can be reached from these references. It looks at each reference in turn, and if non-null, the GC knows that the object it refers to is reachable. There may be duplicates—multiple roots may refer to the same object, so the GC keeps track of which objects it has already seen. For each newly-discovered object, the GC adds all of the instance fields of reference type in that object to the list of references it needs to look at, again, discarding any duplicates. (This includes any hidden fields generated by the compiler, such as those for automatic properties, which I described in Chapter 3.) This means that if an object is reachable, so are all the objects to which it holds references. The GC repeats this process until it runs out of new references to examine. Any objects that it has *not* discovered to be reachable are therefore unreachable, because the GC is simply doing what the program does: a program can only use objects that are accessible either directly or indirectly through its variables, temporary local storage, static fields, and other roots.

Going back to Example 7-1, what would all this mean if the CLR decides to run the GC when we construct the `WebClient`? The `fullUri` variable is still live so the `Uri` it refers to is reachable, but the `baseUri` is no longer live. We did pass a copy of `baseUri` into the constructor for the second `Uri`, and if that had held onto a copy of the reference, then it wouldn't matter that `baseUri` is not live—as long as there's some way to get to an object by starting from a root reference, then the object is reachable. But as it happens, the second `Uri` won't do that, so the first `Uri` the example allocates would be

deemed to be unreachable, and the CLR would be free to recover the memory it had been using.

One important upshot of how reachability is determined is that the GC is unfazed by circular references. This is one reason .NET uses GC instead of reference counting (which is COM's approach). If you have two objects that refer to each other, a reference counting scheme will consider both objects to be in use because each is referred to at least once. But the objects may be unreachable—if there are no other references to the objects, the application will not have any way to use them. Reference counting fails to detect this, so it could cause memory leaks, but with the scheme used by the CLR's GC, the fact that they refer to each other is irrelevant—the garbage collector will never get to either of them, so it will correctly determine that they are no longer in use.

Accidentally Defeating the Garbage Collector

Although the GC can discover ways that your program could reach an object, it has no way to prove that it necessarily will. Take the impressively idiotic piece of code in Example 7-2. Although you'd never write code this bad, it makes a common mistake. It's a problem that usually crops up in more subtle ways, but I want show it in a more obvious example first. Once I've shown how it prevents the GC from freeing objects that we're not going to be using, I'll describe a less straightforward but more realistic scenario in which this same problem often occurs.

Example 7-2. An appallingly inefficient piece of code

```
static void Main(string[] args)
{
    var numbers = new List<string>();
    long total = 0;
    for (int i = 1; i < 100000; ++i)
    {
        numbers.Add(i.ToString());
        total += i;
    }
    Console.WriteLine("Total: {0}, average: {1}",
        total, total / numbers.Count);
}
```

This adds together the numbers from 1 to 100,000 and then prints their average. The first mistake here is that we don't even need a loop, because there's a simple and very well-known closed-form solution for this sort of sum: `n*(n+1)/2`, with `n` being 100,000 in this case. That mathematical gaffe notwithstanding, this does something even more stupid: it builds up a list containing every number it adds, but all it does with that list is to retrieve its `Count` property to calculate an average at the end. Just to make things worse, the code converts each number into a string before putting it in the list. It never actually uses those strings.

Obviously this is a contrived example. That said, I wish I could say I'd never encountered anything this bafflingly pointless in real programs. Sadly, I've come across genuine examples at least this bad, although they were all better obfuscated—when you encounter this sort of thing in the wild, it normally takes half an hour or so to work out that it really is doing something as staggeringly pointless as this. However, my point here is not to lament standards of software development. The purpose of this example is to show how you can run into a shortcoming of the garbage collector.

Suppose the loop in Example 7-2 has been running for a while—perhaps it's on its 90,000th iteration, and is trying to add an entry to the `numbers` list. Suppose that the `List<string>` has used up its spare capacity and the `Add` method will therefore need to allocate a new, larger internal array. The CLR may decide at this point to run the garbage collector to see if it can free up some space. What will happen?

Example 7-2 creates three kinds of objects: it constructs a `List<string>` at the start, it creates a new `string` each time round the loop by calling `ToString()` on an `int`, and more subtly, it will cause the `List<string>` to allocate a `string[]` to hold references to those strings, and because we keep adding new items, it will have to allocate larger and larger arrays. (That array is an implementation detail of `List<string>`, so we can't see it directly.) So the question is: which of these objects will the GC be able to discard to make space for a larger array in the call to `Add`?

Our program's `numbers` variable remains live until the final line of the program, and we're looking at an earlier point in the code, so the `List<string>` object is reachable. The `string[]` array object it is currently using must also be reachable: it's allocating a newer larger one, but it will need to copy the contents of the old one across to the new one so the list must still have a reference to that current array stored in one of its fields. So that array is still reachable, which in turn means that every string element the array refers to will also be reachable. Our program has created 90,000 strings so far, and the garbage collector will find all of them by starting at our `numbers` variable, looking at the fields of the `List<string>` object that refers to, and then looking at every element in the array that one of the list's private fields refers to.

The only allocated items that the GC might be able to collect are old `string[]` arrays that the `List<string>` created back when the list was smaller, and which it no longer has a reference to. By the time we've added 90,000 items, the list will probably have resized itself quite a few times. So depending on when the GC last ran, it will probably be able to find a few of these now-unused arrays. But more interesting here is what it cannot free.

The program never uses the 90,000 strings it creates, so ideally, we'd like the GC to free up the memory they occupy—they will be taking up a few megabytes. We can see very easily that these strings are not used because this is such a short program. But the GC will not know that—it bases its decisions on reachability, and it correctly determines that all 90,000 strings are reachable by starting at the `numbers` variable. And as far as the GC is concerned, it's entirely possible that the list's `Count` property, which we use after the loop finishes, will look at the contents of the list. You and I happen to know that it won't because it doesn't need to, but that's because we know what the `Count` property means. For the GC to infer that our program will never use any of the list's elements directly or indirectly, it would need to know what `List<string>` does inside its `Add` and `Count` methods. This would mean analysis with a level of detail far beyond the mechanisms I've described, which could make garbage collections considerably more expensive. Moreover, even with the serious step up in complexity required to detect which reachable objects this example will never use, in more realistic scenarios the GC would be unlikely to be able to make predictions that were any better than relying on reachability alone.

For example, a much more plausible way to run into this problem is in a cache. If you write a class that caches data that is expensive to fetch or calculate, imagine what would happen if your code only ever added items to the cache, and never removed them. All of the cached data would be reachable for as long as the cache object itself is reachable. The

problem is that your cache will consume more and more space, and unless your computer has sufficient memory to hold every piece of data that your program could conceivably need to use, it will eventually run out of memory.

A naïve developer might complain that this is supposed to be the garbage collector's problem. The whole point of GC is meant to be that I don't need to think about memory management, so why am I running out of memory all of a sudden? But of course the problem is that the GC has no way of knowing which objects it's safe to remove. It is not clairvoyant, so it cannot accurately predict which cached items your program may need in the future—if the code is running in a server, future cache usage could depend on what requests the server receives, something the GC cannot predict. So although it's possible to imagine memory management smart enough to analyze something as simple as Example 7-2, in general this is not a problem the GC can solve. So if you add objects to collections and you keep those collections reachable, the GC will treat everything in those collections as being reachable. It's your job to decide when to remove items.

Collections are not the only situation in which you can fool the garbage collector. As I'll show in Chapter 9, there's a common scenario in which careless use of events can cause memory leaks. More generally, if your program makes it possible for an object to be reached, the GC has no way of working out whether you're going to use that object again, so it has to be conservative.

Having said that, there is a technique for mitigating this with a little help from the GC.

Weak References

Although the garbage collector will follow ordinary references in a reachable object's fields, it's possible to hold a *weak reference*. The garbage collector does not follow weak references, so if the only way to reach an object is through weak references, the garbage collector behaves as though the object is not reachable, and will remove it. A weak reference provides a way to tell the CLR: don't keep this object around on my account, but for as long as something else needs it, I'd like to be able to get access to it.

There are two classes for managing weak references. `WeakReference<T>` is new to .NET 4.5. If you're using an older version of .NET, you'll need to use the non-generic `WeakReference`. The newer class takes advantage of generics to provide a cleaner API than the original, which was introduced back in .NET 1.0 before generics came along. In fact the newer class has a somewhat different API. I'll show that first, and then I'll talk about the older class. Example 7-3 shows a cache that uses `WeakReference<T>`.

Example 7-3. Using weak references in a cache

```
public class WeakCache<TKey, TValue> where TValue : class
{
    private Dictionary<TKey, WeakReference<TValue>> _cache =
        new Dictionary<TKey, WeakReference<TValue>>();

    public void Add(TKey key, TValue value)
    {
        _cache.Add(key, new WeakReference<TValue>(value));
    }

    public bool TryGetValue(TKey key, out TValue cachedItem)
    {
```

```
        WeakReference<TValue> entry;
        if (_cache.TryGetValue(key, out entry))
        {
            bool isAlive = entry.TryGetTarget(out cachedItem);
            if (!isAlive)
            {
                _cache.Remove(key);
            }
            return isAlive;
        }
        else
        {
            cachedItem = null;
            return false;
        }
    }
}
```

This cache stores all values via a `WeakReference<T>`. Its `Add` method simply passes the object to which we'd like a weak reference as the constructor argument for a new `WeakReference<T>`. The `TryGetValue` method attempts to retrieve a value previously stored with `Add`. It first checks to see if the dictionary contains a relevant entry. If it does, that entry's value will be the `WeakReference<T>` we created earlier. My code calls that weak reference's `TryGetTarget` method, which will return true if the object is still available, and false if it has been collected.

> Availability doesn't necessarily imply reachability. The object may have become unreachable since the most recent GC. Or there may not even have been a GC since the object was allocated. `TryGetTarget` doesn't care whether the object is reachable right now, it only cares whether it has been collected yet.

If the object is available, it provides it through an `out` parameter, and the value it provides back will be a strong reference. So if this method returns true, we don't need to worry about any race condition in which the object becomes unreachable moments later—the fact that we've now stored that reference in the variable the caller supplied via the `cachedItem` reference will keep the target alive. If `TryGetTarget` returns false, my code removes the relevant entry from the dictionary, because it represents an object that no longer exists. Example 7-4 tries this code out, forcing a couple of garbage collections so we can see it in action.

Example 7-4. Exercising the weak cache

```
var cache = new WeakCache<string, byte[]>();

var data = new byte[100];
cache.Add("d", data);

byte[] fromCache;
Console.WriteLine("Retrieval: " + cache.TryGetValue("d", out fromCache));
Console.WriteLine("Same ref?  " + object.ReferenceEquals(data, fromCache));
fromCache = null;

GC.Collect();
Console.WriteLine("Retrieval: " + cache.TryGetValue("d", out fromCache));
Console.WriteLine("Same ref?  " + object.ReferenceEquals(data, fromCache));
```

```
fromCache = null;

data = null;
GC.Collect();
Console.WriteLine("Retrieval: " + cache.TryGetValue("d", out fromCache));
Console.WriteLine("Non-null?  " + (fromCache != null));
```

This begins by creating an instance of my cache class, and then adding a reference to a 100-byte array to the cache. It also stores a reference to the same array in a local variable called `data`, which remains live until its final usage near the bottom of the code, in which I set its value to `null`. The example tries to retrieve the value from the cache immediately after adding it, and also uses `object.ReferenceEquals` just to check that the value we get back really refers to the same object that we put in. Then I force a garbage collection, and try again. (This sort of artificial test code is one of the very few situations in which you'd want to do this—see the "Forcing Garbage Collection" section later for details.) Since the `data` variable still holds a reference to the array, and is still live, the array is still reachable, so we would expect the value still to be available from the cache. Next I set `data` to `null`, so my code is no longer keeping that array reachable. The only remaining reference is a weak one, so when I force another GC, we expect the array to be collected, so the final lookup in the cache should fail. To verify this, I check both the return value, expecting false, and the value returned through the `out` parameter, which should be `null`. And that is exactly what happens when running the program, as you can see:

```
Retrieval: True
Same ref?  True
Retrieval: True
Same ref?  True
Retrieval: False
Non-null?  False
```

If you're using an older version of .NET (v4.0 or earlier), you'll need to use the non-generic `WeakReference` class to create a weak reference. Its constructor also takes a reference to the object to which you'd like to maintain a weak reference. However, retrieving the reference works slightly differently. This class provides an `IsAlive` property, which will return false if the GC has determined that the object is no longer reachable. Note that if it returns true, that's no guarantee that the object is still reachable. This property merely tells you whether the object has been collected by the GC yet.

The `WeakReference`'s `Target` property returns a reference to the object. (This property is of type `object`, because this is the non-generic version, so you'll need to cast it.) This returns a strong reference (i.e., a normal one) so if you store this in either a local variable, or a field of a reachable object, or if you merely use its value in an expression, that will have the effect of making the object reachable again, so you do not need to worry about the object being removed in between you retrieving the reference from `Target` and using it. However, there is a race condition between `IsAlive` and `Target`: it's entirely possible that in between testing the `IsAlive` property, and reading `Target`, a garbage collection could occur, meaning that although `IsAlive` returned true, the object is no longer available. `Target` returns null if the object has gone, so you should always test for that. `IsAlive` is only interesting if you want to discover whether an object has gone but don't actually want to do anything with it if it's still there. (For example, if you have a collection containing weak references, you might periodically want to purge all of the entries whose objects are no longer alive.)

> The generic `WeakReference<T>` does not provide an `IsAlive` property. This avoids a potential misuse that can arise with the non-generic version. An easy mistake to make would be to test the `IsAlive` property, and then just assume that if it returns true, `Target` will necessarily return a non-null value. If a GC happens at exactly the wrong moment, that won't be true. The generic version avoids this problem by forcing you to use the atomic `TryGetValue` method. If you want to test for availability without using the target, just call `TryGetValue` and then don't use the reference it returns.

Later, I will describe finalization, which complicates matters by introducing a twilight zone in which the object has been determined to be unreachable, but has not yet gone. Objects that are in this state are typically of little use, so by default, a weak reference (either generic or non-generic) will treat objects waiting for finalization as though they have already gone. This is called a *short weak reference*. If for some reason you need to know whether an object has really gone (rather than merely being on its way out), both weak reference classes have constructor overloads, some of which can create a *long weak reference*, which provides access to the object even in this zone between unreachability and final removal.

Reclaiming Memory

So far, I've described how the CLR determines which objects are no longer in use, but not what happens next. Having identified the garbage, the runtime must then collect it. The CLR uses slightly different strategies for small and large objects. (It defines a large object as one bigger than 85,000 bytes.) Most allocations involve small objects, so I'll write about those first.

The CLR tries to keep the heap's free space contiguous. Obviously that's easy when the application first starts up, because there's nothing but free space, and it can keep things contiguous by allocating memory for each new object directly after the last one. But after the first garbage collection occurs, the heap is unlikely to look so neat. Most objects have short lifetimes, and it's common for the majority of objects allocated after any one GC to be unreachable by the time the next GC runs. However, some will still be in use. From time to time, applications create objects that hang around for longer, and of course whatever work was in progress when the GC ran will probably be using some objects, so the most recently-allocated heap blocks are likely still to be in use. This means that the end of the heap might look something like Figure 7-1, where the grey rectangles are the reachable blocks, while the white ones show blocks that are no longer in use. (In practice, the GC would not normally kick in until you had allocated a lot more blocks than this. An accurate diagram would be more cluttered, but would otherwise look similar.)

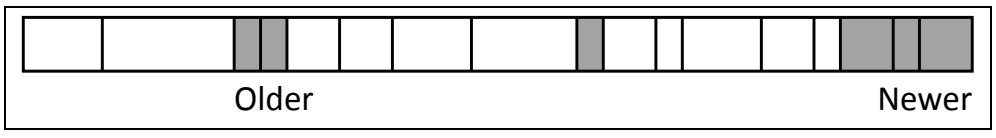

Figure 7-1. Section of heap with some reachable objects

One possible allocation strategy would be to start using these empty blocks as new memory is required, but there would be a couple of problems with that. First, it tends to be wasteful, because the blocks the application requires will probably not fit precisely

into the holes available. Second, finding a suitable empty block can be somewhat expensive, particularly if there are lots of gaps and you're trying to pick one that will minimize waste. It's not impossibly expensive of course—lots of heaps work this way—but it's a lot more costly than the initial situation where each new block could be allocated directly after the last one because all the spare space was contiguous. The expense of heap fragmentation is non-trivial, so the CLR typically tries to get the heap back into a state where the free space is contiguous. As Figure 7-2 shows, it moves all the reachable objects towards the start of the heap, so that all the free space is at the end, which puts it back in the favorable situation of being able to allocate new heap blocks one after another in the contiguous lump of free space.

> The runtime has to ensure that any references to these relocated blocks continue to work after the blocks have moved. The CLR happens to implement references as pointers (although the CLI spec does not require this—a reference is just a value that identifies some particular instance on the heap). It already knows where all the references to any particular block are because it had to find them to discover which blocks were reachable. It adjusts all these pointers when it moves the block.

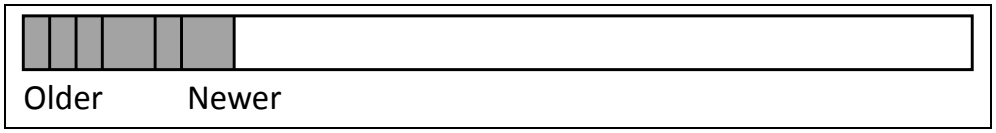

Figure 7-2. Section of heap after compaction

Besides making heap block allocation a relatively cheap operation, compaction offers another performance benefit. Because blocks are allocated into a contiguous area of free space, objects that were created in quick succession will typically end up right next to each other in the heap. This is significant because the caches in modern CPUs tend to favor locality, i.e., they perform best when related pieces of data are stored close together.

The low cost of allocation and the high likelihood of good locality can sometimes mean that garbage collected heaps offer better performance than traditional heaps that require the program to free memory explicitly. This may seem surprising, given that the GC appears to do a lot of extra work that is unnecessary in a non-collecting heap. Some of that 'extra work' is illusory—something has to keep track of which objects are in use, and traditional heaps just push that housekeeping overhead into our code. However, relocating existing memory blocks comes at a price, so the CLR uses some tricks to minimize the amount of copying it needs to do.

The older an object is, the more expensive it will be for the CLR to compact the heap once it finally becomes unreachable. If the most recently allocated object is unreachable when the GC runs, compaction is free for that object: there are no more objects after it so nothing needs to be moved. Compare that with the very first object your program allocates—if that becomes unreachable, compaction would mean moving every reachable object on the heap. More generally, the older an object is, the more objects will be sat after it, so the more data will need to be moved to compact the heap. Copying 20MB of data to save 20 bytes does not sound like a great tradeoff. So the CLR will often defer compaction for older parts of the heap.

To decide what counts as 'old' the CLR divides the heap into *generations*. The boundaries between generations move around at each GC, because generations are defined in terms of how many GCs an object has survived. Any object that was allocated after the previous GC is in generation 0, because it has not yet survived any collections. When the GC next runs, any generation 0 objects that are still reachable will be moved as necessary to compact the heap, and will then be deemed to be in generation 1.

Objects in generation 1 are not yet considered to be old. The GC typically runs while the code is right in the middle of doing things—after all, it runs when space on the heap is being used up, and that won't happen if the program is idle. So there's a high chance that some of the recently-allocated objects represent work in progress, and will become unreachable shortly. Generation 1 acts a sort of holding zone while we wait to see which objects are short-lived and which are longer lived.

As the program continues to execute, the GC will run from time to time, promoting new objects that survive into generation 1. Some of the objects in generation 1 will become unreachable. However, the GC does not necessarily compact this part of the heap immediately—it may allow a few generation 0 collections and compactions in between each generation 1 compaction, but it will happen eventually. Objects that survive this stage are moved into generation 2, which is the oldest generation.

The CLR attempts to recover memory from generation 2 much less frequently than from other generations. Years of research and analysis have shown that in most applications, objects that survive into generation 2 are likely to remain reachable for a long time, so if they do eventually become unreachable, it's likely that the object is very old, and so will the objects around it be. This means that compacting this part of the heap to recover the memory is costly for two reasons: not only will this old object probably be followed by a large number of other objects, requiring a large volume of data to be copied, the memory it occupied might not have been used for a long time, meaning it's probably no longer in the CPU's cache, slowing down the copy even further. And the caching costs will continue after collection, because if the CPU has had to shift megabytes of data around in old areas of the heap, this will probably have the side effect of cleaning out the CPU's cache—cache sizes can be as small as 512KB at the very low-power, low-cost end of the spectrum, and can be 30MB more in high-end server-oriented chips, but in the mid-range anything from 2MB to 16MB of cache is typical, and many .NET applications' heaps will be larger than that. Most of the data the application had been using would have been in the cache right up until the generation 2 GC, but would be gone once the GC has finished. So when the GC completes, and normal execution resumes, the code will run in slow motion for a while until the data the application needs is loaded back into the cache.

Generations 0 and 1 are sometimes referred to as the *ephemeral* generations, because they mostly contain objects that exist only for a short while. The contents of these parts of the heap will often be in the CPU's cache because they will have been accessed recently, so compaction is not particularly expensive for these sections. Moreover, because most objects have a short lifetime, the majority of memory that the garbage collector is able to collect will be from objects in these first two generations, so these are likely to offer the greatest reward (in terms of memory recovered) in exchange for the CPU time expended. So it's common to see several ephemeral collections a second in a busy program, but it's also common for several minutes to elapse between each generation 2 collection.

The CLR has another trick up its sleeve for generation 2 objects. They often don't change very much, so there's a high likelihood that during the first phase of a GC, in which the runtime detects which objects are reachable, it would be repeating some work it did

earlier, because it will follow exactly the same references and will produce the same results for significant subsections of the heap. So the CLR will sometimes use the operating system's memory protection services to detect when older heap blocks are modified. This can enable it to rely on summarized results from earlier GC operations instead of having to redo all of the work every time.

How does the GC decide whether to collect just from generation 0, rather than 1 or even 2? Collections for all three generations are triggered by using up a certain amount of memory. So for generation 0 allocations, once you have allocated some particular number of bytes since the last GC, a new GC will occur. The objects that survive this will move into generation 1, and the CLR keeps track of the number of bytes added to generation 1 since the last generation 1 collection, and if that exceeds a threshold, generation 1 will be collected too. Generation 2 works in the same way. The thresholds are not documented, and in fact they're not even constant—the CLR monitors your allocation patterns and modifies these thresholds to try and find a good balance for making efficient use of memory, minimizing the CPU time spent in the GC, and avoiding the excessive latency that could arise if the CLR waited a very long time between collections, leaving huge amounts of work to do when the collection finally occurs.

> This explains why, as mentioned earlier, the CLR doesn't necessarily wait until it has actually run out of memory before triggering a GC. It may be more efficient to run one sooner.

You may be wondering how much of the preceding information is useful. After all, the bottom line would appear to be that the CLR ensures that heap blocks are kept around for as long as they are reachable, and that some time after they become unreachable it will eventually reclaim their memory, and it employs a strategy designed to do this efficiently. Are the details of this generational optimization scheme relevant to a developer? They are, insofar as they tell us that some coding practices are likely to be more efficient than others.

The most obvious upshot of the process is that the more objects you allocate, the harder the GC will have to work. But you'd probably guess that without knowing anything about the implementation. More subtly, larger objects cause the GC to work harder—collections for each generation are triggered by the amount of memory your application uses. So bigger objects don't just increase memory pressure, they also end up consuming more CPU cycles as a result of triggering more frequent GCs.

Perhaps the most important fact to emerge from an understanding of the generational collector is that the length of an object's lifetime has an impact on how hard the garbage collector has to work. Objects that live for a very short time are handled very efficiently, because the memory they use will be recovered quickly in a generation 0 or 1 collection, and the amount of data that needs to be moved to compact the heap will be small. Objects that live for an extremely long time are also OK because they will end up in generation 2. They will not to be moved about very often, because collections are infrequent for that part of the heap. Furthermore, the CLR may be able to use the Windows memory manager's write detection feature to manage reachability discovery for old objects more efficiently. However, although very short-lived and very long-lived objects are handled efficiently, objects that live long enough to get into generation 2 but not much longer are a problem. Microsoft occasionally describes this occurrence as a *mid-life crisis*.

If your application has a lot of objects making it into generation 2 which go on to become unreachable, the CLR will need to perform collections on generation 2 more often than it

otherwise might. (In fact, generation 2 is only collected during a *full collection*, which also collects free space previously used by large objects.) These are usually significantly more expensive than other collections. Compaction requires more work with older objects, but also, more housekeeping is required when disrupting the generation 2 heap—the picture the CLR has built up about reachability within this section of the heap may need to be rebuilt, and the GC will need to disable the write detection used to enable that while it compacts the heap, which incurs a cost. There's a good chance that most of this part of the heap will not be in the CPU's cache either, so working with it can be slow.

Full garbage collections consume significantly more CPU time than collections in the ephemeral generations. In user interface applications, this can cause delays long enough to be irritating for the user, particularly if parts of the heap had been paged out. In server applications, full collections may cause significant blips in the typical time taken to service a request. Such problems are not the end of the world of course, and as I'll describe later, recent versions of the CLR have made significant improvements in this area. Even so, minimizing the number of objects that survive to generation 2 is good for performance. You would need to consider this when designing code that caches interesting data in memory—a cache aging policy that failed to take the GC's behavior into account could easily behave inefficiently and if you didn't know about the perils of middle-aged objects, it would be hard to work out why. Also, as I'll show later in this chapter, the mid-life crisis issue is one reason you might want to avoid C# destructors where possible.

I have left out some heap operation details by the way. For example, I've not talked about how the GC typically dedicates sections of the address space to the heap in fixed-size chunks, nor the details of how it commits and releases memory. Interesting though these mechanisms are, they have much less relevance to how you design your code than an awareness of the assumptions that a generational garbage collector makes about typical object lifetimes.

There's one last thing to talk about on the topic of collecting memory from unreachable objects. As mentioned earlier, large objects work differently. There's a separate heap called, appropriately enough, the *Large Object Heap* (LOH) and the CLR uses this for any object larger than 85,000 bytes. That's just the object itself, and not the sum total of all the memory an object allocates during construction. An instance of the `GreedyObject` class in Example 7-5 would be tiny—it only needs enough space for a single reference, plus the heap block overhead. In a 32-bit process, that would be 4 bytes for the reference and 8 bytes of overhead, and double that in a 64-bit process. However, the array to which it refers is 400,000 bytes long, so that would go on the LOH, while the `GreedyObject` itself would go on the ordinary heap.

Example 7-5. A small object with a large array

```
public class GreedyObject
{
    public int[] MyData = new[100000];
}
```

It's technically possible to create a class whose instances are large enough to require the LOH, but it's unlikely to happen outside of generated code or highly contrived examples. In practice, most LOH heap blocks will contain arrays.

The biggest difference between the LOH and the ordinary heap is that the GC does not compact the LOH, because copying large objects is expensive. It works more like a traditional C heap: the CLR maintains a list of free blocks, and decides which block to

use based on the size requested. However, the list of free blocks is populated by the same unreachability mechanism as is used by the rest of the heap.

Garbage Collector Modes

Although the CLR will tune some aspects of the GC's behavior at runtime (e.g, by dynamically adjusting the thresholds that trigger collections for each generation), it offers several modes designed to suit different kinds of applications. These fall into two broad categories: workstation and server, but there are variations within each. Workstation is the default. To configure server mode, you will need an application configuration file. (Web applications usually have one by default, called *web.config*. Outside of the ASP.NET web framework, the configuration file is normally called *app.config*, and many Visual Studio project templates provide this file automatically.) Example 7-6 shows a configuration file that enables server GC mode.

Example 7-6. Configuring server GC

```
<?xml version="1.0" ?>
<configuration>
     <runtime>
        <gcServer enabled="true" />
     </runtime>
</configuration>
```

The workstation modes are designed, predictably enough, for the workloads that client-side code typically has to deal with, in which the process is usually working on either one task, or a small number of tasks at any one time. Workstation mode offers two variations: non-concurrent and concurrent. The non-concurrent mode is designed to optimize throughput on a single processor with a single core. In fact, this is the only option on such machines—neither concurrent workstation mode nor server mode is available on such hardware. But where multiple logical processors are available, the workstation GC defaults to concurrent mode. (If for some reason you want to disable concurrent mode on a multi-processor machine, you can add `<gcConcurrent enabled="false" />` inside the `<runtime>` element of your configuration file.)

In concurrent mode, the GC minimizes the amount of time for which it suspends threads during a garbage collection. There are certain phases of the GC in which the CLR has to suspend execution to ensure consistency, and for collections from the ephemeral generations, threads will be suspended for the majority of the operation. This is usually fine because these collections normally run very quickly—they take a similar amount of time to a page fault that didn't cause any disk activity. (These non-blocking page faults happen fairly often on Windows, and are fast enough that a lot of developers seem to be unaware that they even occur.) Full collections are the problem, and it's these that the concurrent mode handles differently.

For client-side code, the greatest concern is to avoid delays long enough to be visible to users. The purpose of concurrent GC is to enable code to continue to execute while some parts of the collection occur. Not all of the work done in a collection really needs to bring everything to a halt. To maximize opportunities for concurrency, concurrent mode uses more memory than the non-concurrent mode, and also reduces overall throughput, but for interactive applications, that's usually a good tradeoff if the perceived performance improves. Users are more sensitive to delays in response than they are to suboptimal average utilization of the CPU.

As well as concurrent collection, you will also see Microsoft's documentation refer to background collection. This is not a separate mode, and there is no distinct setting because it's just something that happens in concurrent workstation mode. Background collection is an enhancement, introduced in .NET 4.0, that addresses a specific shortcoming of concurrent collection: although threads can continue to run while a full collection is performed, prior to .NET 4.0 they would come to a halt if they used up their generation 0 quota—the GC was unable to start an ephemeral collection until the full collection had finished, so if the application allocated memory while a concurrent GC was in progress, things could still grind to a halt. The background collection feature fixes this by allowing ephemeral collections to occur without waiting for the full collection to complete, and it also allows the heap to grow by allocating more memory from the OS while a background GC is in progress. This means the GC is more often able to deliver on the promise of minimizing interruptions.

Server mode is significantly different than workstation mode. It is only available when you have multiple logical processors (e.g., a multi-core CPU, or multiple physical CPUs). Its availability has nothing to do with which version of Windows you're running by the way—server mode is available on non-server editions and server editions alike if you have suitable hardware, and workstation mode is always available. Each processor gets its own section of the heap, so when a thread is working on its own problem independently of the rest of the process, it can allocate heap blocks with minimal contention. In server mode, the CLR creates several threads dedicated to garbage collection work, one for each logical CPU in the machine. These run with higher priority than normal threads, so when garbage collections do occur, all available CPU cores go to work on their own heaps, which can provide better throughput than workstation mode with large heaps.

> Objects created by one thread can still be accessed by others—logically, the heap is still a unified service. Server mode is just an implementation strategy that is optimized for workloads where all the threads work on their own jobs mostly in isolation. It also works best if the jobs all have similar heap allocation patterns.

There are some problems that can arise with server mode. It works best when only one process on the machine uses this mode, because it is set up to try to use all CPU cores simultaneously during collections, and it also tends to use considerably more memory than workstation mode. If a single server hosts multiple .NET processes that all do this, contention for resources could reduce efficiency. Another issue with server GC is that it favors throughput over response time. In particular, collections happen less frequently, because this tends to increase the throughput benefits that multi-CPU collections can offer, but it also means that each individual collection takes longer.

The duration of a full collection in server mode can create problems on applications with large heaps—it can cause serious delays in responsiveness on a web site, for example. There are a couple of ways you can mitigate this. You can request notifications shortly before the collection occurs (using the `GC` class's `RegisterForFullGCNotification`, `WaitForFullGCApproach` and `WaitForFullGCComplete` methods), and if you have a server farm, a server that's running a full GC may be able to ask the load balancer to avoid passing it requests until the GC completes. Alternatively, with .NET 4.5 or later, you can use background collection—in .NET 4.0, concurrent background collections were only available in workstation mode, but .NET 4.5 adds these to server mode. Since background collections

allow application threads to continue to run and even to perform generation 0 and 1 collections while the full collection proceeds in the background, it significantly improves the application's response time during collections, while still delivering the throughput benefits of server mode.

Accidentally Defeating Compaction

Heap compaction is an important feature of the CLR's garbage collector, because it has a strong positive impact on performance. Certain operations can prevent compaction, and that's something you'll want to minimize, because fragmentation can increase memory use and reduce performance significantly.

To be able to compact the heap, the CLR needs to be able to move heap blocks around. Normally it can do this because it knows all of the places in which your application refers to heap blocks, and it can adjust all the references when it relocates a block. But what if you're calling a Windows API that works directly with the memory you provide? For example, if you read data from a file or a network socket, how will that interact with garbage collection?

If you use system calls that read or write data using devices such as the disk or network interface, these normally work directly with your application's memory. If you read data from the disk, the operating system will typically instruct the disk controller to put the bytes directly into the memory your application passed to the API. The OS will perform the necessary calculations to translate the virtual address into a physical address. (Virtual memory means that the value your application puts in a pointer is only indirectly related to the actual address in your computer's RAM.) The OS will lock the pages into place for the duration of the IO request, to ensure that the physical address remains valid. It will then supply the disk system with that address. This enables the disk controller to copy data from the disk directly into memory, without needing any further involvement from the CPU. This is very efficient, but runs into problems when it encounters a compacting heap. What if the block of memory is a `byte[]` array on the heap? Suppose a GC occurs in between us asking to read the data, and the disk being able to supply the data. (If it's a mechanical disk with spinning platters, it can take 10ms or more to start supplying data, which is an age in CPU terms, so the chances are fairly high.) If the GC were to decide to relocate our `byte[]` array to compact the heap, the physical memory address that the OS gave to the disk controller would be out of date, so when the controller started putting data into memory, it would be writing to the wrong place. At best it would put the bytes into what is now some free space at the end of the heap, but it could well overwrite some unrelated object that's using the space previously occupied by the `byte[]` array.

There are three ways the CLR could deal with this. One would be to make the GC wait—heap relocations could be suspended while I/O operations are in progress. But that's a non-starter—a busy network server can run for days without ever entering a state in which no I/O operations are in progress. In fact, the server doesn't even need to be busy. It might allocate several `byte[]` arrays to hold the next few incoming network requests, and would typically try to avoid getting into a state where it didn't have at least one such buffer available. The OS would have pointers to all of these and may well have supplied the network card with the corresponding physical address so that it can get to work the moment data starts to arrive. So even if the server is idle, it still has certain buffers that cannot be relocated.

An alternative would be for the CLR to provide a separate non-moving heap for these sorts of operations. Perhaps we could allocate a fixed block of memory for an I/O operation, and then copy the results into the `byte[]` array on the GC heap once the I/O has finished. But that's also not a brilliant solution. Copying data is expensive—the more copies you make of incoming or outgoing data, the slower your server will run, so you really want network and disk hardware to copy the data directly to or from its natural location. And if this hypothetical fixed heap was more than an implementation detail of the CLR, if it was available for application code to use directly to minimize copying, that might open the door to all the memory management bugs that garbage collection is supposed to banish.

So the CLR uses a third approach: it selectively prevents heap block relocations. The garbage collector is free to run while I/O operations are in progress, but certain heap blocks can be *pinned*. Pinning a block sets a flag that tells the GC that the block cannot currently be moved. So if the garbage collector encounters such a block, it will simply leave it where it is, but will attempt to relocate everything around it.

There are three ways C# code normally causes heap blocks to be pinned. You can do so explicitly using the `fixed` keyword. This allows you to obtain a raw pointer to a storage location such as a field or an array element, and the compiler will generate code that ensures that for as long as a fixed pointer is in scope, the heap block to which it refers will be pinned. A more common way to pin a block is through interop, i.e., calls into unmanaged code, such as a method on a COM component, or a Win32 API. If you make an interop call to an API that requires a pointer to something, the CLR will detect when that points to a heap block, and it will automatically pin the block. (By default, the CLR will unpin it automatically when the method returns. If you're calling an asynchronous API, you can use the `GCHandle` class mentioned earlier to pin a heap block until you explicitly unpin it.) I will describe interop and raw pointers in Chapter 23.

The third and most common way to pin heap blocks is also the least direct: many class library APIs call unmanaged code on your behalf, and will pin the arrays you pass in as a result. For example, the class library defines a `Stream` class that represents a stream of bytes. There are several implementations of this abstract class. Some streams work entirely in memory, but some wrap I/O mechanisms, providing access to files, or to the data being sent or received through a network socket. The abstract `Stream` base class defines methods for reading and writing data via `byte[]` arrays, and the I/O-based stream implementations will often pin the heap blocks containing those arrays for as long as necessary.

If you are writing an application that does a lot of pinning (e.g., a lot of network I/O) you may need to think carefully about how you allocate the arrays that get pinned. Pinning does the most harm for recently-allocated objects, because these live in the area of the heap where most compaction activity occurs. Pinning recently allocated blocks tends to cause the ephemeral section of the heap to fragment. Memory that would normally have been recovered almost instantly must now wait for blocks to become unpinned, meaning that by the time the collector can get to those blocks, a lot more other blocks will have been allocated after them, meaning that a lot more work is required to recover the memory.

If pinning is causing your application problems there will be a few common symptoms. The percentage of CPU time spent in the GC will be relatively high—anything over 10% is considered to be bad. But that alone does not necessarily implicate pinning—it could be the result of middle-aged objects causing too many full collections. So you can

monitor the number of pinned blocks on the heap[2] to see if these are the specific culprit. If it looks like excessive pinning is causing you pain, there are two ways to avoid this. One is to design your application so that you only ever pin blocks that live on the large object heap. Remember, the LOH is not compacted, so pinning does not impose any cost—the GC wasn't going to move the block in any case. The challenging part of this is that it forces you to do all of your I/O with arrays that are at least 85,000 bytes long. That's not necessarily a problem because most I/O APIs can be told to work with a subsection of the array. So if you actually wanted to work with 4,096 byte blocks, you could create one array large enough to hold at least 21 of those blocks. You'd need to write some code to keep track of which slots in the array were in use, but if it fixes a performance problem, it may be worth the effort.

> If you choose to mitigate pinning by attempting to use the LOH, you need to remember that it is an implementation detail. Future versions of .NET could conceivably change the threshold of what constitutes a large object, and they could even remove the LOH entirely. So you'd need to look at this aspect of your design for each new release of .NET.

The other way to minimize the impact of pinning is to try to ensure that pinning mostly happens only to objects in generation 2. If you allocate a pool of buffers and reuse them for the duration of the application, this will mean that you're pinning blocks that the GC is fairly unlikely to want to move, keeping the ephemeral generations free to be compacted at any time. The earlier you allocate the buffers the better, because the older an object is, the less likely the GC is to want to move it.

Forcing Garbage Collection

The `GC` class provides a `Collect` method that allows you to force a garbage collection to occur. You can pass a number indicating the generation you would like to collect, and the overload that takes no arguments performs a full collection. You will very rarely have any good reason to call `GC.Collect`. I'm mentioning it here because it comes up a lot on the web, which could easily make it seem more useful than it is.

Forcing a GC can cause problems. The GC monitors its own performance and tunes its behavior in response to your application's allocation patterns. But to do this, it needs to allow enough time between collections to get an accurate picture of how its current settings are working. If you force collections to occur too often, it will not be able to tune itself, and the outcome will be twofold: the garbage collector will run more often than necessary, and when it does run its behavior will be suboptimal. Both problems are likely to increase the amount of CPU time spent in the GC.

So when would you force a collection? If you happen to know that your application has just finished some work, and is about to go idle, it might be worth considering forcing a collection. Garbage collections are triggered by activity, so if you know that your application is about to go to sleep—perhaps it's a service that has just finished running a batch job, and will not do any more work for another few hours—you know that it won't

[2] The Performance Monitor tool built into Windows can report numerous useful statistics for garbage collection and other CLR activities, including the percentage of CPU time spent in the GC, the number of pinned objects, and the number of generation 0, 1, and 2 collections.

be allocating any new objects and will therefore not trigger the GC automatically. So forcing a GC would provide an opportunity to return memory to the operating system before going to sleep. That said, if this is your scenario, it might be worth looking at mechanisms that would enable your process to exit entirely—Windows provides various ways in which jobs or services that are only required from time to time can be unloaded completely when they are inactive. But if that technique is inapplicable for some reason—perhaps your process has high startup costs, or needs to stay running to receive incoming network requests—a forced full collection might be the next best option.

It's worth being aware that there is one way that a GC can be triggered without your application needing to do anything. When the system is running low on memory, Windows broadcasts a message to all running processes. The CLR handles this message, and forces a garbage collection when it occurs. So even if your application does not proactively attempt to return memory, memory might be reclaimed eventually if something else in the system needs it.

Destructors and Finalization

The CLR performs a lot of work on our behalf to find out when our objects are no longer in use. It's possible to get it to notify you of this—instead of simply removing unreachable objects, the CLR can first tell an object that it is about to be removed. The CLR calls this finalization, but C# presents it through a special syntax: to exploit finalization you must write a destructor.

> If your background is in C++, do not be fooled by the name. As you will see, a C# destructor is a different than a C++ destructor in some important ways.

Example 7-7 shows the syntax for a destructor. This code compiles into an override of a method called `Finalize`, which as Chapter 6 mentioned, is a special method defined by the `object` base class. Finalizers are required always to call the base implementation of `Finalize` that they override. C# generates that call for us to prevent us from violating the rule, which is why it doesn't let us simply write a `Finalize` method directly. Finalizers are not invoked directly—they are called by the CLR, so we do not specify an accessibility level for the destructor.

Example 7-7. Class with destructor

```
public class LetMeKnowMineEnd
{
    ~LetMeKnowMineEnd()
    {
        Console.WriteLine("Goodbye, cruel world");
    }
}
```

The CLR does not guarantee to run finalizers on any particular schedule. First of all, it needs to detect that the object has become unreachable, which won't happen until the GC runs. If your program is idle, that might not happen for a long time—the GC only runs either when your program is doing something, or if system-wide memory pressure causes the GC to spring into life. It's entirely possible that minutes, hours, or even days could pass between your object becoming unreachable and the CLR noticing that it has become unreachable.

Even when the CLR does detect unreachability it still doesn't guarantee to call the finalizer straight away. Finalizers run on a dedicated thread, and this finalization thread runs with low priority, meaning that it will only run when the system has at least one logical CPU with nothing better to do. Also, since there's only one finalization thread, regardless of which GC mode you choose, a slow finalizer will cause other finalizers to wait.

In most cases, the CLR doesn't even guarantee to run finalizers at all. When a process exits, the runtime will wait for a short while for finalizers to complete, but if the finalization thread hasn't managed to run all extant finalizers within two seconds of the program trying to finish, it just gives up and exits anyway. (As the "Critical Finalizers" section later in this chapter explains, there are certain exceptions, but the majority of finalizers get no guarantees.)

In summary, finalizers can be delayed indefinitely if your program is either idle or busy, and are not guaranteed to run. But it gets worse—you can't actually do very much that is useful in a finalizer.

You might think that a finalizer would be a good place to ensure that certain work is properly completed. For example, if your object writes data to a file, but buffers that data so as to be able to write a small number of large chunks rather than writing in tiny dribs and drabs (because large writes are often more efficient), you might think that finalization is the obvious place to ensure that any data in your buffers has been safely flushed out to disk. But think again.

During finalization, an object cannot trust any of the other objects it has references to. If your object's destructor runs, your object must have become unreachable. This means it's highly likely that any other objects yours refers to have also become unreachable. The CLR is likely to discover the unreachability of groups of related objects simultaneously—if your object created three or four objects to help it do its job, the whole lot will become unreachable at the same time. The CLR makes no guarantees about the order in which it runs finalizers (except for critical finalizers which, as I'll explain later, get some weak guarantees). This means it's entirely possible that by the time your destructor runs, all the objects you were using have already been finalized. So if they also perform any last minute cleanup, it's too late to use them. For example, the `FileStream` class, which derives from `Stream` and provides access to a file, closes its file handle in its destructor. So if you were hoping to flush your data out to the `FileStream`, it's too late—the file stream may well already be closed.

So destructors are of remarkably little use: you can have no idea if or when they will run, and you can't use other objects inside a destructor. So what use are they?

> To be fair, although the CLR does not guarantee to run most finalizers, it will usually run them in practice. The absence of guarantees only matters in relatively extreme situations so they're not quite as bad as I've made them sound. Even so, this doesn't mitigate the fact that you cannot, in general, rely on other objects in your destructor.

The only reason finalization exists at all is to make it possible to write .NET types that are wrappers for the sorts of entities that are traditionally represented by handles—things like files, and sockets. These are created and managed outside of the CLR—files and sockets require the operating system kernel to allocate resources; libraries may also provide handle-based APIs, and they will typically allocate memory on their own private

heaps to store information about whatever the handle represents. The CLR cannot see these activities—all it sees is a .NET object with a field containing an integer, and it has no idea that the integer is a handle for some resource outside of the CLR. So it doesn't know that it's important that the handle be closed when the object falls out of use. This is where finalizers come in—they are a place to put code that tells the system that's external to the CLR that the entity represented by the handle is no longer in use. The inability to use other objects is not a problem in this scenario.

> If you are writing code that wraps a handle, you should normally use one of the built-in classes that derive from `SafeHandle` described in Chapter 23, or if absolutely necessary, derive your own. This base class extends the basic finalization mechanism with some handle-oriented helpers, but it also uses the critical finalization mechanism discussed later to guarantee that the finalizer will run. Furthermore, it gets special handling from the interop layer to avoid premature freeing of resources.

It's possible to use finalization for diagnostic purposes, although you should not rely on it, because of the unpredictability and unreliability already discussed. Some classes contain a finalizer which does nothing other than check that the object had not been abandoned in a state where it had unfinished work. For example, if you had written a class that buffers data before writing it to a file as described above, you would need to define some method that callers should use when they are done with your object (perhaps called `Flush` or `Close`), and you could write a finalizer that checks to see if the object was put into a safe state before being abandoned, raising an error if not. This would provide a way to discover when programs had forgotten to clean things up correctly. (The .NET Framework's Task Parallel Library, which I'll describe in Chapter 17, uses this technique. When an asynchronous operation throws an exception, it uses a finalizer to discover when the program that launched it fails to get around to detecting that exception.)

If you write a finalizer, you should disable it when your object is in a state where it no longer requires finalization, because finalization has its costs. If you offer a `Close` or `Flush` method, finalization is unnecessary once these have been called, so you should call the `GC` class's `SuppressFinalize` class to let the GC know that your object no longer needs to be finalized. If your object's state subsequently changes, you can call the `ReRegisterForFinalize` method to re-enable it.

The greatest cost of finalization is that it guarantees that your object will survive at least into the first generation and possibly longer. Remember, all objects that survive from generation 0 make it into generation 1. If your object has a finalizer, and you have not disabled it by calling `SuppressFinalize`, the CLR cannot get rid of your object until it has run its finalizer. And since finalizers run asynchronously on a separate thread, the object has to remain alive even though it has been found to be unreachable. So the object is not yet collectable, even though it is unreachable. It therefore lives on into generation 1. It will usually be finalized shortly afterwards, meaning that the object will then become a waste of space until a generation 1 collection occurs. Those happen rather less frequently than generation 0 collections. If your object had already made it into generation 1 before becoming unreachable, a finalizer increases the chances of getting into generation 2 just before it is about to fall out of use. A finalized object therefore makes inefficient use of memory, which is a reason to avoid finalization, and a reason to disable it whenever possible in objects that do sometimes require it.

> Even though `SuppressFinalize` can save you from the most egregious costs of finalization, an object that uses this technique still has higher overheads than an object with no finalizer at all. The CLR does some extra work when constructing finalizable objects to keep track of which have not yet been finalized. (Calling `SuppressFinalize` just takes your object back out of this tracking list.) So although suppressing finalization is much better than letting it occur, it's better still if you don't ask for it in the first place.

A slightly weird upshot of finalization is that an object that the GC discovered was unreachable can make itself reachable again. It's possible to write a destructor that stores the `this` reference in a root reference, or perhaps in a collection that is reachable via a root reference. Nothing stops you from doing this, and the object will continue to work (although its finalizer will not run a second time if the object becomes unreachable again) but it's a slightly odd thing to do. This is referred to as *resurrection*, and just because you can do it doesn't mean you should. It is best avoided.

Critical Finalizers

Although in general, finalizers are not guaranteed to run, there are exceptions: you can write a *critical finalizer*. A finalizer is critical if and only if it belongs to a class that derives from the `CriticalFinalizerObject` base class. The CLR makes two useful guarantees for objects of this kind. First, the CLR will give the finalizer an opportunity to run, even in situations where the usual time limit for finalization has been exhausted. Second, within any group of objects that were discovered to be unreachable at the same time, the CLR will run critical finalizers after it has finished running non-critical ones, meaning that if you write a finalizable object with a reference to an object with a critical finalizer, it is safe to use that object in your own finalizer.

The CLR disallows certain operations inside critical finalizers. They are not allowed to construct new objects or throw exceptions, and they can only invoke methods if those methods follow the same constraints. These constraints mean the CLR can still guarantee to run critical finalizers even in relatively extreme situations, such as when shutting down a process due to low memory. The constraints prevent you from using critical finalization as a general-purpose mechanism for overcoming the limitations of ordinary finalization. It is a highly constrained mechanism designed to make it possible to close handles reliably.

Earlier, I mentioned the `SafeHandle` class, which is the preferred way to wrap handles in .NET. It can guarantee to free handles because it derives from `CriticalFinalizerObject`. If you rely on this class or one of the classes derived from it to ensure your handles get freed, your own classes may not need to derive from `CriticalFinalizerObject`, so your own finalizer would not be subject to the critical finalization constraints. Also, because of the ordering guarantees, you could be sure that a handle wrapped in a `SafeHandle` will still be valid when your finalizer runs because the critical finalizer in `SafeHandle` won't have run yet. Better yet, by using a `SafeHandle`, you may be able to get away without needing write your own finalizer at all.

I hope that by now, I have convinced you that destructors do not provide a useful general purpose mechanism for shutting down objects cleanly. They are mostly only useful for

dealing with handles for things that live outside of the CLR's control. If you need timely, reliable cleanup of resources, there's a better mechanism.

IDisposable

The class library defines an interface called `IDisposable`. The CLR does not treat this interface as being in any way special, but C# has some built-in support for it. `IDisposable` is a very simple abstraction—as Example 7-8 shows, it defines just one member, the `Dispose` method.

Example 7-8. The IDisposable interface

```
public interface IDisposable
{
    void Dispose();
}
```

The idea behind `IDisposable` is very simple. If your code uses an object that implements this interface, you should call `Dispose` once you have finished using that object. This provides the object with an opportunity to free up any resources it may have allocated. If it was using resources represented by handles, it will typically close those handles immediately rather than waiting for finalization to kick in (and would suppress finalization at the same time). If the object was using services on some remote machine in a stateful way—perhaps holding a connection open to a server to be able to make requests—it would immediately let the remote system know that it no longer requires the services, in whatever way is necessary (e.g., by closing the connection).

> There is a persistent myth that calling `Dispose` causes the garbage collector to do something. You may read on the web that `Dispose` finalizes the object, or even that it causes the object to be garbage collected. This is nonsense. The CLR does not handle `IDisposable` or `Dispose` differently than any other interface or method.

`IDisposable` is important because it's possible for an object to consume very little memory, and yet to tie up some expensive resources. For example, consider an object that represents a connection to a database. Such an object might not need many fields—it could even have just a single field containing a handle representing the connection. From the CLR's point of view this is a pretty cheap object, and we could allocate hundreds of the things without triggering a garbage collection. But in the database server, things would look different—it might need to allocate a considerable amount of memory for each incoming connection. Connections might even be strictly limited by licensing terms. (This illustrates that 'resource' is a fairly broad concept—it means pretty much anything that you might run out of.)

Relying on garbage collection to notice when database connection objects are no longer in use is likely to be a bad strategy. The CLR will know that we've allocated, say, 50 of the things, but if that only consumes a few hundred bytes in total, it will see no reason to run the GC. And yet our application may be about to grind to a halt—if we only have 50 connection licenses for the database, the next attempt to create a connection will fail. And even if there's no licensing limitation, we could still be making highly inefficient use of database resources by opening far more connections than we need.

It's imperative that we close connection objects as soon as we can, without waiting for the GC to tell us which ones are out of use. This is where `IDisposable` comes in. It's not just for database connections, of course. It's critically important for any object that is a front for something that lives outside of the CLR such as a file or a network connection. Even for resources that are not especially constrained, `IDisposable` provides a way to tell objects when we are finished with them so they can shut down cleanly, solving the problem described earlier for the object that performs internal buffering.

> If a resource is expensive to create, you will typically want to reuse it. This is often the case with database connections, so the usual practice is to maintain a pool of connections. Instead of closing a connection when you're finished with it, you return it to the pool, making it available for reuse. (.NET's data access APIs can do this for you, as I'll show in Chapter 19.) The `IDisposable` model is still useful here. When you ask a resource pool for a resource, it usually provides a wrapper around the real resource, and when you dispose that wrapper, it returns the resource to the pool instead of freeing it. So calling `Dispose` is really just a way of saying "I'm done with this" and it's up to the `IDisposable` implementation to decide what to do next.

Implementations of `IDisposable` are required to tolerate multiple calls to `Dispose`. Although this means consumers can call `Dispose` multiple times without harm, they should not attempt to use an object after it has been disposed. In fact, the class library defines a special exception that objects can throw if you misuse them in this way: `ObjectDisposedException`. (I will discuss exceptions in Chapter 8.)

You're free to call `Dispose` directly of course, but C# also supports `IDisposable` in two ways: `foreach` loops and `using` statements. A `using` statement is a way to ensure that you reliably dispose an object that implements `IDisposable` once you're done with it. Example 7-9 shows how to use it.

Example 7-9. A using statement

```
using (FileStream reader = File.OpenText(@"C:\temp\File.txt"))
{
    Console.WriteLine(reader.ReadToEnd());
}
```

This is equivalent to the code in Example 7-10. The `try` and `finally` keywords are part of C#'s exception handling system, which I'll discuss in detail in Chapter 8. In this case, they're being used to ensure that the code inside the `finally` block executes even if something goes wrong in the code inside the `try` block. This also ensures that `Dispose` gets called even if you execute a `return` statement in the middle of the block, or even use the `goto` statement to jump out of it.

Example 7-10. How using statements expand

```
{
    FileStream reader = File.OpenText(@"C:\temp\File.txt"))
    try
    {
        Console.WriteLine(reader.ReadToEnd());
    }
    finally
    {
```

```
        if (reader != null)
        {
            ((IDisposable) reader).Dispose();
        }
    }
}
```

If the `using` statement's variable type is a value type, C# will not generate the code that checks for `null`, and will just invoke `Dispose` directly.

If you need to use multiple disposable resources within the same scope, you can stack multiple `using` statements in front of a single block. Example 7-11 uses this to copy the contents of one file to another.

Example 7-11. Stacking using statements

```
using (Stream source = File.OpenRead(@"C:\temp\File.txt"))
using (Stream copy = File.Create(@"C:\temp\Copy.txt"))
{
    source.CopyTo(copy);
}
```

Stacking of `using` statements is not a special syntax. It's just an upshot of the fact that a `using` statement is always followed by single embedded statement which will be executed before `Dispose` gets called. Normally that statement is a block, but in Example 7-11, the first `using` statement's embedded statement is the second `using` statement.

A `foreach` loop generates code that will use `IDisposable` if the enumerator implements it. Example 7-12 shows a `foreach` loop that uses just such an enumerator.

Example 7-12. A foreach loop

```
foreach (string file in Directory.EnumerateFiles(@"C:\temp"))
{
    Console.WriteLine(file);
}
```

The `Directory` class's `EnumerateFiles` method returns an `IEnumerable<string>`. As you saw in Chapter 5, this has a `GetEnumerator` method that returns an `IEnumerator<string>`, an interface which inherits from `IDisposable`. Consequently, the C# compiler will produce code equivalent to Example 7-13.

Example 7-13. How foreach loops expand

```
{
    IEnumerator<string> e =
        Directory.EnumerateFiles(@"C:\temp").GetEnumerator();
    try
    {
        while (e.MoveNext())
        {
            string file = e.Current;
            Console.WriteLine(file);
        }
    }
    finally
    {
```

```
        if (e != null)
        {
            ((IDisposable) e).Dispose();
        }
    }
}
```

There are a few variations the compiler can produce, depending on the collection's enumerator type. If it's a value type that implements `IDisposable`, the compiler won't generate the check for `null` in the `finally` block (just like in a `using` statement). If the static type of the enumerator does not implement `IDisposable`, the outcome depends on whether the type is open for inheritance. If it is sealed, or if it is a value type, the compiler will not generate code that attempts to call `Dispose` at all. If it is not sealed, the compiler generates code in the `finally` block that tests at runtime whether the enumerator implements `IDisposable`, and then calls `Dispose` if it does, and does nothing otherwise.

> Although Example 7-13 represents how C# 5.0 compiles `foreach` loops, I should point out that earlier versions of the compiler did something subtly different. (It doesn't affect `IDisposable` handling. I mention it here for completeness.) Notice that the iteration variable, `file`, is declared inside the `while` loop, so each iteration effectively gets a new variable. This used to be declared before the `while` loop, so there was one variable used throughout, and its value changed with each iteration. Most of the time, this makes no discernible difference, but in Chapter 9, we'll see a scenario in which this matters.

The `IDisposable` interface is easiest to consume when you obtain a resource and finish using it in the same method, because you can write a `using` statement (or where applicable, a `foreach` loop) to ensure that you call `Dispose`. But sometimes, you will write a class which creates a disposable object and puts a reference to it in a field, because it needs to be able to use that object over a longer timescale. For example, you might write a logging class, and if a logger object writes data to a file, it might hold onto a `FileStream` object. C# provides no automatic help here, so it's up to you to ensure that any contained objects get disposed. You would write your own implementation of `IDisposable` which disposed the other objects. As Example 7-14 shows, this is not rocket science. Note that this example sets `_file` to `null`, so it will not attempt to dispose the file twice. This is not strictly necessary, because the `FileStream` will tolerate multiple calls to `Dispose`. But it does give the `Logger` object an easy way to know that it is in a disposed state, so if we were to add some real methods, we could check `_file` and throw an `ObjectDisposedException` if it is `null`.

Example 7-14. Disposing a contained instance

```
public sealed class Logger : IDisposable
{
    private FileStream _file;

    public Logger(string filePath)
    {
        _file = File.CreateText(filePath);
    }
```

```
    public void Dispose()
    {
        if (_file != null)
        {
            _file.Dispose();
            _file = null;
        }
    }
    // A real class would go on to do something with the FileStream of course
}
```

This example dodges an important problem. The class is sealed, which avoids the issue of how to cope with inheritance. If you write an unsealed class that implements `IDisposable`, you should provide a way for a derived class to add its own disposal logic. The most straightforward solution would be to make `Dispose` virtual so that a derived class can override it, performing its own cleanup in addition to calling your base implementation. However, there is a slightly more complicated pattern that you will see from time to time in .NET.

Some objects implement `IDisposable` and also have a finalizer. Since the introduction of `SafeHandle` and related classes in .NET 2.0, it's relatively unusual for a class to need to provide both (unless it derives from `SafeHandle`). Only wrappers for handles normally need finalization, and classes that use handles now typically defer to a `SafeHandle` to provide that, rather than implementing their own finalizers. However, there are exceptions, and some library types implement a pattern designed to support both finalization and `IDisposable`, allowing you to provide custom behaviors for both in derived classes. For example, the `Stream` base class works this way.

The pattern is to define a protected overload of `Dispose` that takes a single `bool` argument. The base class calls this from its public `Dispose` method and also its destructor, passing in true or false respectively. That way, you only have to override one method, the protected `Dispose`. It can contain any logic common to both finalization and disposal, such as closing handles, but you can also perform any disposal-specific or finalization-specific logic because the argument tells you which sort of cleanup is being performed. Example 7-15 shows how this might look.

Example 7-15. Custom finalization and disposal logic

```
public class MyFunkyStream : Stream
{
    // For illustration purposes only. Usually better
    // to use some type derived from SafeHandle.
    private IntPtr _myCustomLibraryHandle;
    private Logger _log;

    protected override void Dispose(bool disposing)
    {
        base.Dispose(disposing);

        if (_myCustomLibraryHandle != IntPtr.Zero)
        {
            MyCustomLibraryInteropWrapper.Close(_myCustomLibraryHandle);
            _myCustomLibraryHandle = IntPtr.Zero;
        }
        if (disposing)
        {
```

```
            if (_log != null)
            {
                _log.Dispose();
                _log = null;
            }
        }
    }

    ... overloads of Stream's abstract methods would go here
}
```

This hypothetical example is a custom implementation of the `Stream` abstraction that uses some external non-.NET library that provides handle-based access to resources. We want to close the handle when the public `Dispose` method is called, but if that hasn't happened by the time our finalizer runs, we want to close the handle then. So the code checks to see if the handle is still open and closes it if necessary, and it does this whether the call to the `Dispose(bool)` overload happened as a result of the object being explicitly disposed, or being finalized—we need to ensure that the handle is closed in either case. However, this class also appears to use an instance of the `Logger` class from Example 7-14. Because that's an ordinary object, we shouldn't attempt to use it during finalization, so we only attempt to dispose it if our object is being disposed. If we are being finalized, then although `Logger` itself is not finalizable, it uses a `FileStream`, which is finalizable, and it's quite possible that the `FileStream` finalizer will already have run by the time our `MyFunkyStream` class's finalizer runs, so it would be a bad idea to call methods on the `Logger`.

When a base class provides this virtual protected form of `Dispose`, it should call `GC.SuppressFinalization` in its public `Dispose`. The `Stream` base class does this. More generally, if you find yourself writing a class that offers both `Dispose` and a finalizer, then whether or not you choose to support inheritance with this pattern, you should in any case suppress finalization when `Dispose` is called.

Boxing

While I'm discussing garbage collection and object lifetime, there's one more topic I should talk about in this chapter: boxing. Boxing is the process that enables a variable of type `object` to refer to a value type. An `object` variable is only capable of holding a reference to something on the heap, so how can it refer to an `int`? What happens when the code in Example 7-16 runs?

Example 7-16. Using an int as an object

```
class Program
{
    static void Show(object o)
    {
        Console.WriteLine(o.ToString());
    }

    static void Main(string[] args)
    {
        int num = 42;
        Show(num);
    }
```

```
}
```

The `Show` method expects an object, and I'm passing it `num`, which is a local variable of the value type `int`. In these circumstances, C# generates a box, which is essentially a reference type wrapper for a value. The CLR can automatically provide a box for any value type, although if it didn't, you could write something that does the same job—Example 7-17 shows a hand-built box.

Example 7-17. Not actually how a box works

```
// Not a real box, but similar in effect.
public class Box<T>
    where T : struct
{
    public readonly T Value;
    public Box(T v)
    {
        Value = v;
    }

    public override string ToString()
    {
        return Value.ToString();
    }

    public override bool Equals(object obj)
    {
        return Value.Equals(obj);
    }

    public override int GetHashCode()
    {
        return Value.GetHashCode();
    }
}
```

This is a fairly ordinary class that contains a single instance of a value type as its only field. If you invoke the standard members of `object` on the box, this class's overrides make it look as though you invoked them directly on the field itself. So if I passed `new Box<int>(num)` as the argument to `Show` in Example 7-16, I would be asking to construct a new `Box<int>`, copying the value of `num` into the box, and `Show` would receive a reference to that box. When `Show` called `ToString`, the box would call the `int` field's `ToString`, so you'd expect the program to print out 42.

In fact, we don't need to write Example 7-17, because the CLR will build the box for us. It will create an object on the heap that contains a copy of the boxed value, and which forwards the standard `object` methods to the boxed value. And it does something that we can't. If you ask a box its type by calling `GetType`, it will return the same type object as you'd get if you called `GetType` directly on an `int` variable—I can't do that with my custom `Box<T>` because `GetType` is not virtual. Also, getting back the underlying value is easier than it would be with a hand-built box, because unboxing is an intrinsic CLR feature.

If you have a reference of type `object`, and you cast it to `int`, the CLR checks to see if the reference does indeed refer to a boxed `int`, and if it does, it returns a copy of the boxed value. So inside the `Show` method of Example 7-16, I could write `(int) o` to get

back a copy of the original value whereas if I were using the class in Example 7-17, I'd need the more convoluted `((Box<int>) o).Value`.

Boxes are automatically available for all structs, not just the built-in value types. If the struct implements any interfaces, the box will provide all the same interfaces. (That's another trick that Example 7-17 cannot perform.)

Some implicit conversions cause boxing. You can see this in Example 7-16—I have passed an expression of type `int` where `object` was required, without needing an explicit cast. Implicit conversions also exist between a value and any of the interfaces that value's type implements. For example, you can assign a value of type `int` into a variable of type `IComparable<int>` without needing a cast. This causes a box to be created, because variables of any interface type are like variables of type `object`: they can only hold a reference to an item on the garbage collected heap.

Implicit boxing can occasionally cause problems for either of two reasons. First, it makes it easy to generate extra work for the garbage collector—the CLR does not make any attempt to cache boxes, so if you write a loop that executes 100,000 times, and that loop contains an expression that uses an implicit boxing conversion, you'll end up generating 100,000 boxes, which the GC will have to clean up just like anything else on the heap. Second, each box operation (and each unbox) copies the value, which might not provide the semantics you were expecting. Example 7-18 illustrates some potentially surprising behavior.

Example 7-18. Illustrating the pitfalls of mutable structs

```
public struct DisposableValue : IDisposable
{
    private bool _disposedYet;

    public void Dispose()
    {
        if (!_disposedYet)
        {
            Console.WriteLine("Disposing for first time");
            _disposedYet = true;
        }
        else
        {
            Console.WriteLine("Was already disposed");
        }
    }
}

class Program
{
    static void CallDispose(IDisposable o)
    {
        o.Dispose();
    }

    static void Main(string[] args)
    {
        var dv = new DisposableValue();
        Console.WriteLine("Passing value variable:");
        CallDispose(dv);
        CallDispose(dv);
```

```
        CallDispose(dv);

        IDisposable id = dv;
        Console.WriteLine("Passing interface variable:");
        CallDispose(id);
        CallDispose(id);
        CallDispose(id);

        Console.WriteLine("Calling Dispose directly on value variable:");
        dv.Dispose();
        dv.Dispose();
        dv.Dispose();
    }
}
```

The `DisposableValue` struct implements the `IDisposable` interface we saw earlier. It keeps track of whether it has been disposed already. The program contains a method that calls `Dispose` on any `IDisposable` instance. The program declares a single variable of type `DisposableValue` and passes this to `CallDispose` three times. Here's the output from that part of the program:

```
Passing value variable:
Disposing for first time
Disposing for first time
Disposing for first time
```

On all three occasions, the struct seems to think this is the first time we've called `Dispose` on it. That's because each call to `CallDispose` created a new box—we are not really passing the `dv` variable, we are passing a newly boxed copy each time, so the `CallDispose` method is working on a different instance of the struct each time. This is consistent with how value types normally work—even when they're not boxed, when you pass them as arguments, you end up passing a copy (unless you use the `ref` keyword).

The next part of the program ends up generating just a single box—it assigns the value into another local variable of type `IDisposable`. This uses the same implicit conversion as we did when passing the variable directly as an argument, so this creates yet another box, but it only does so once, and we then pass the same reference to this particular box three times over, which explains why the output from this phase of the program looks different:

```
Passing interface variable:
Disposing for first time
Was already disposed
Was already disposed
```

These three calls to `CallDispose` all use the same box, which contains an instance of our struct, and so after the first call it remembers that it has been disposed already. Finally, our program calls `Dispose` directly on the local variable, producing this output:

```
Calling Dispose directly on value variable:
Disposing for first time
Was already disposed
Was already disposed
```

No boxing is involved at all here, so we are modifying the state of the local variable. Someone who only glanced at the code might not have expected this output—we have already passed the `dv` variable to a method that called `Dispose` on its argument, so it

might be surprising to see that it thinks it hasn't been disposed first time round. But once you understand that `CallDispose` requires a reference and therefore cannot use a value directly, it's clear that every call to `Dispose` before this point has operated on some boxed copy, and not the local variable. (Obviously if we were to pass `dv` as an argument to `CallDispose` again, we'd expect it to say it was already disposed, because that call would generate yet another boxed copy, but this time, we're copying a value that's already in the state of having been disposed.)

The behavior is all straightforward when you understand what's going on, but it requires you to be mindful that you're dealing with a value type, and to understand when boxing causes implicit copying. This is one of the reasons Microsoft discourages developers from writing value types that can change their state—if a value cannot change, then a boxed value of that type also cannot change. It matters less whether you're dealing with the original or a boxed copy, so there's less scope for confusion.

Boxing used to be a much more common occurrence in early versions of .NET. Before generics arrived in .NET 2.0, collection classes all worked in terms of `object`, so if you wanted a resizable list of integers, you'd end up with a box for each `int` in the list. Generic collection classes do not cause boxing—a `List<int>` is able to store unboxed values directly.

Summary

In this chapter, I described the heap that the runtime provides. I showed the strategy that the CLR uses to determine which heap objects can still be reached by your code, and the generation-based mechanism it uses to reclaim the memory occupied by objects that are no longer in use. The garbage collector is not clairvoyant, so if your program keeps an object reachable, the GC has to assume that you might use that object in the future. This means you will sometimes need to be careful to make sure you don't cause memory leaks by accidentally keeping hold of objects for too long. We looked at the finalization mechanism, and its various limitations and performance issues, and we also looked at `IDisposable`, which is the preferred system for cleaning up non-memory resources. Finally, we saw how value types can act like reference types thanks to boxing.

In the next chapter, I will show how C# presents the CLR's error handling mechanisms.

8

Exceptions

Some operations can fail. If your program is reading data from a file stored on an external drive, someone might disconnect the drive. Your application might try to construct an array only to discover that the system does not have enough free memory. Intermittent wireless network connectivity can cause network requests to fail. One widely used way for a program to discover these sorts of failures is for each API to return a value indicating whether the operation succeeded. This requires developers to be vigilant if all errors are to be detected, because programs must check the return value of every operation. This is certainly a viable strategy, but it can obscure the code—the logical sequence of work to be performed when nothing goes wrong can get buried by all of the error checking, making the code harder to maintain. C# supports another popular error handling mechanism that can avoid this problem: exceptions.

When an API reports failure with an exception, this disrupts the normal flow of execution, leaping straight to the nearest suitable error handling code. This enables a degree of separation between error handling logic and the code that tries to perform the task at hand. This can make code easier to read and maintain, although it does have the downside of making it harder to see all the possible ways in which the code may execute.

Exceptions can also report problems with operations where a return code might not be practical. For example, the runtime can detect and report problems for various operations, even something as simple as using a reference. Reference type variables can contain `null`, and if you try to invoke a method on a null reference, it will fail. The runtime reports this with an exception.

Most errors in .NET are represented as exceptions. However, some APIs offer you a choice between return codes and exceptions. For example, the `int` type has a `Parse` method that takes a string and attempts to interpret its contents as a number, and if you pass it some non-numeric text (e.g., `"Hello"`) it will indicate failure by throwing a `FormatException`. If you don't like that you can call `TryParse` instead, which does exactly the same job, but if the input is non-numeric, it simply returns `false` instead of throwing an exception. (Since the method's return value has the job of reporting success or failure, the method provides the integer result via an `out` parameter). Numeric parsing is not the only operation to use this pattern, in which a pair

of methods (`Parse` and `TryParse` in this case) provide a choice between exceptions and return values. As you saw in Chapter 5, dictionaries offer a similar choice. The indexer throws an exception if you use a key that's not in the dictionary, but you can also look up values with `TryGetValue`, which returns false on failure, just like `TryParse`. Although this pattern crops up in a few places, for the majority of APIs exceptions are the only choice.

If you are designing an API that could fail, how should it report failure? Should you use exceptions, a return value, or both? Microsoft's class library design guidelines contain instructions that seem unequivocal:

> "Do not return error codes. Exceptions are the primary means of reporting errors in frameworks."

But how does that square with the existence of `int.TryParse`? The guidelines have a section on performance considerations for exceptions that says this:

> "Consider the TryParse pattern for members that may throw exceptions in common scenarios to avoid performance problems related to exceptions."

Failing to parse a number is not necessarily an error. For example, you might want your application to allow the month to be specified numerically or as text. So there are certainly common scenarios in which the operation might fail, but the guideline has another criterion: you should only offer the `TryParse` approach when the operation is fast compared to the time taken to throw and handle an exception.

Exceptions can typically be thrown and handled in a fraction of a millisecond, so they're not desperately slow—nothing like as slow as reading data from disk for example, but they're not blindingly fast either. I find that on my computer, a single thread can parse five-digit numeric strings at a rate of roughly ten million strings per second, and it's capable of rejecting non-numeric strings at about the same speed if I use `TryParse`. The `Parse` method handles numeric strings just as fast, but it's about 400 times slower at rejecting non-numeric strings than `TryParse`, thanks to the cost of exceptions. Of course, converting strings to integers is a pretty fast operation, so this makes exceptions look particularly bad, but that's why this pattern is most common on operations that are naturally fast.

> Exceptions can be particularly slow when debugging. This is partly because the debugger has to decide whether to break in, but it's particularly pronounced with the first unhandled exception your program hits in Visual Studio's debugger. This can give the impression that exceptions are considerably more expensive than they really are. The numbers in the preceding paragraph are based on observed runtime behavior without debugging overheads. Having said that, those numbers slightly understate the costs, because handling an exception tends to cause the CLR to run bits of code and access data structures it would not otherwise need to use, which can have the effect of pushing useful data out of the CPU's cache. This can cause code to run slower for a short while after the exception has been handled, until the non-exceptional code and data can make its way back into the cache.

Most APIs do not offer a `Try`*`Xxx`* form, and will report all failures as exceptions, even in cases where failure might be common. For example, the file APIs do not provide a way to open an existing file for reading without throwing an exception if the file is missing.

(You can use a different API to test whether the file is there first, but that's no guarantee of success. It's always possible for some other process to delete the file between you asking whether it's there and attempting to open it.) Since filesystem operations are inherently slow, the `TryXxx` pattern would not offer a worthwhile performance boost here even though it might make logical sense.

Exception Sources

Class library APIs are not the only source of exceptions. They can be thrown in any of the following scenarios:

- Your program uses a class library API, which detects a problem
- Your own code detects a problem
- The runtime detects the failure of an operation, e.g. arithmetic overflow in a checked context, or an attempt to use a null reference, or an attempt to allocate an object for which there is not enough memory
- The runtime detects a situation outside of your control that affects your code, e.g., your thread is being aborted due to application shutdown

Although these all use the same exception handling mechanisms, the places in which the exceptions emerge are different. I'll describe where to expect each sort of exception in the following sections.

Exceptions from APIs

With an API call, there are several kinds of problems that could result in exceptions. You may have provided arguments that make no sense, such as a null reference where a non-null one is required, or an empty string where the name of a file was expected. Or the arguments might look OK individually, but not collectively. For example, you could call an API that copies data into an array, asking it to copy more data than will fit. You could describe these as 'that will never work' style errors, and they are usually the result of mistakes in the code.

A subtly different class of problems arises when the arguments all look plausible, but the operation turns out not to be possible given the current state of the world—for example, you might ask to open a particular file but the file may not be present, or perhaps it exists but some other program already has it open and has demanded exclusive access to the file. Yet another variation is that things may start well, but conditions can change, so perhaps you opened a file successfully and have been reading data for a while, but then the file becomes inaccessible. As suggested earlier, someone may have unplugged a disk, or the drive could have failed due to overheating or age.

Asynchronous programming adds yet another variation. In Chapters 17 and 18, I'll show various asynchronous APIs—ones where work can progress after the method that started it has returned. Work that progresses asynchronously can also fail asynchronously, in which case the library might have to wait until your code next calls into it before it can report the error.

Despite the variations, in all these cases the exception will come from some API that your code calls. (Even with asynchronous errors, exceptions emerge either when you try to

collect the result of an operation, or when you explicitly ask whether an error has occurred.) Example 8-1 shows some code where exceptions of this kind could emerge.

Example 8-1. Getting an exception from a library call

```
static void Main(string[] args)
{
    using (var r = new StreamReader(@"C:\Temp\File.txt"))
    {
        while (!r.EndOfStream)
        {
            Console.WriteLine(r.ReadLine());
        }
    }
}
```

There's nothing categorically wrong with this program, so we won't get any exceptions complaining about arguments being self-evidently wrong. If your computer's *C:* drive has a *Temp* folder, and if that contains a *File.txt* file, and if the user running the program has permission to read that file, and if nothing else on the computer has already acquired exclusive access to the file, and if there are no problems such as disk corruption that could make any part of the file inaccessible, and if no new problems (such as the drive catching fire) develop while the program runs, this code will work just fine: it will print each line of text in the file. But that's a lot of ifs.

If there is no such file, the `StreamReader` constructor will not complete. Instead, it will throw an exception. This program makes no attempt to handle that, so the application would terminate. If you ran the program outside of Visual Studio's debugger, you would see the following output:

```
Unhandled Exception: System.IO.FileNotFoundException: Could not find file
'C:\Temp\File.txt'.
   at System.IO.__Error.WinIOError(Int32 errorCode, String maybeFullPath)
   at System.IO.FileStream.Init(String path, FileMode mode, FileAccess access,
Int32 rights, Boolean useRights, FileShare share, Int32 bufferSize, FileOptions
options, SECURITY_ATTRIBUTES secAttrs, String msgPath, Boolean bFromProxy,
Boolean useLongPath)
   at System.IO.FileStream..ctor(String path, FileMode mode, FileAccess access,
FileShare share, Int32 bufferSize, FileOptions options)
   at System.IO.StreamReader..ctor(String path, Encoding encoding, Boolean
detectEncodingFromByteOrderMarks, Int32 bufferSize)
   at System.IO.StreamReader..ctor(String path)
   at ConsoleApplication1.Program.Main(String[] args) in
C:\dev\ConsoleApplication1\Program.cs:line 13
```

This tells us what error occurred, and it shows the full call stack of the program at the point at which the problem happened. Windows would also show its error reporting dialog, and depending on how your computer is configured, it may even report the crash to Microsoft's error reporting service. If you run the same program in Visual Studio's debugger, that will tell you about the exception, and it will also highlight the line on which the error occurred, as Figure 8-1 shows.

[I will produce a non-ClearType version of this image in due course.]

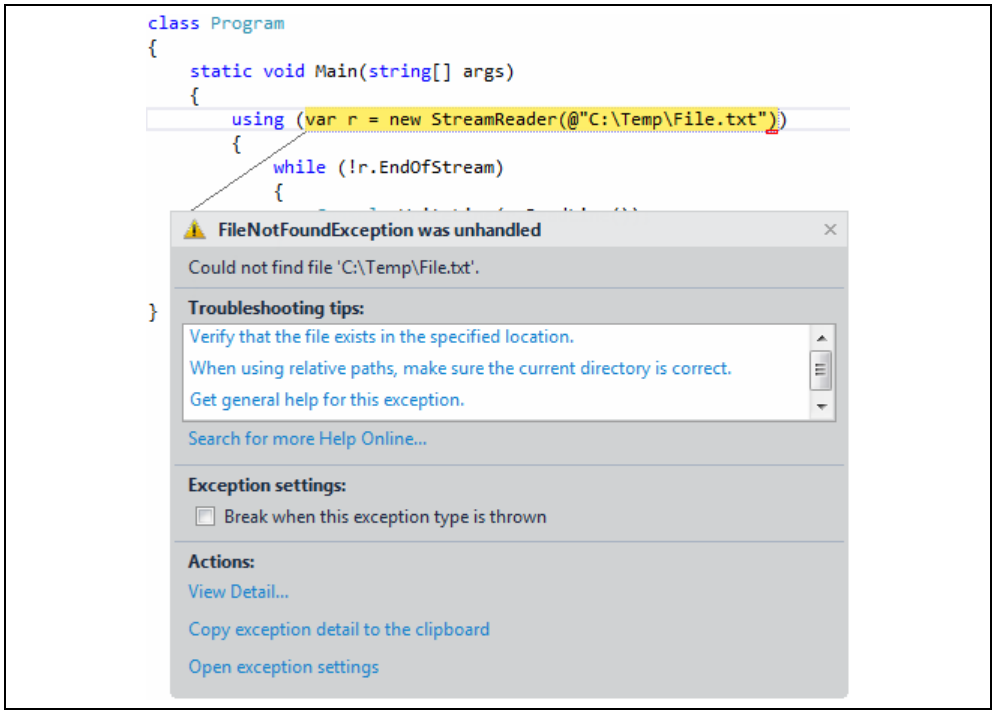

Figure 8-1. Visual Studio reporting an exception

What we're seeing here is the default behavior that occurs when a program does nothing to handle exceptions: if a debugger is attached, it will step in, and if not, the program just crashes. I'll show how to handle exceptions soon, but this illustrates that you cannot simply ignore them.

The call to the `StreamReader` constructor is not the only line that could throw an exception in Example 8-1 by the way. The code calls `ReadLine` multiple times, and any of those calls could fail. In general, any member access could result in an exception, even just reading a property, although class library designers usually try to minimize the extent to which properties throw exceptions. If you make an error of the 'that will never work' kind, then a property might throw an exception, but usually not for errors of the 'this particular operation didn't work' kind. For example, the documentation states that the `EndOfStream` property used in Example 8-1 would throw an exception if you tried to read it after having disposed the `StreamReader` object—an obvious coding error—but if there are problems reading the file, `StreamReader` will only throw exceptions from methods or the constructor.

Exceptions from Your Code

The second potential source of errors mentioned earlier is when your own code detects a problem and decides to throw an exception. I'll be showing examples of that later. For now, I'm just describing where you can expect exceptions to come from, and from that perspective, this sort of exception is fairly similar to ones that emerge from a class library. In fact, class libraries use the same mechanisms for throwing exceptions that you can. When you throw your own exceptions, it will always be clear exactly where in the code exceptions may arise: they will originate on the lines of code from which you

explicitly throw exceptions, and will emerge from the methods that contain those lines of code.

Failures Detected by the Runtime

The third source of exceptions is when the CLR itself detects that some operation has failed. Example 8-2 shows a method in which this could happen. As with Figure 8-1, there's nothing innately wrong with this code (other than not being very useful). It's perfectly possible to use this without causing problems. However, if someone passes in zero as the second argument, the code will attempt an illegal operation.

Example 8-2. A potential runtime-detected failure

```
static int Divide(int x, int y)
{
    return x / y;
}
```

The CLR will detect when this division operation attempts to divide by zero, and will throw a `DivideByZeroException`. This will have the same effect as an exception from an API call: if the program makes no attempt to handle the exception, it will crash, or the debugger will break in.

> Division by zero is not always illegal. Floating point types support special values representing positive and negative infinity, which is what you get when you divide a positive or negative value by zero; if you divide zero by itself, you get the special Not a Number value. None of the integer types support these special values, so integer division by zero is always an error.

The final source of exceptions I described earlier is also the detection of certain failures by the runtime, but they work slightly differently. They are not necessarily triggered directly by anything that your code did on the thread on which the exception occurred. These are sometimes referred to as *asynchronous exceptions*, and they can in theory be thrown at literally any point in your code, making it hard to ensure that you can deal with them correctly. However, these only tend to be thrown in fairly catastrophic circumstances, often when your program is about to be shut down in any case, so only very specialized code needs to deal with these. I will return to them later.

I've described the usual situations in which exceptions are thrown, and you've seen the default behavior, but what if you want your program to do something other than crash?

Handling Exceptions

When an exception is thrown, the CLR looks for code to handle the exception. The default exception handling behavior only comes into play if there are no suitable handlers anywhere on the entire call stack. To provide a handler, we use C#'s `try` and `catch` keywords, as Example 8-3 shows.

Example 8-3. Handling an exception

```
try
{
    using (StreamReader r = new StreamReader(@"C:\Temp\File.txt"))
```

```
    {
        while (!r.EndOfStream)
        {
            Console.WriteLine(r.ReadLine());
        }
    }
}
catch (FileNotFoundException)
{
    Console.WriteLine("Couldn't find the file");
}
```

The block immediately following the `try` keyword is usually called a *try block*, and if the program throws an exception while it's inside such a block, the CLR looks for matching *catch blocks*. Example 8-3 has just a single catch block, and in the parentheses following the `catch` keyword, you can see that this particular block is intended to handle exceptions of type `FileNotFoundException`.

You saw earlier that if there is no *C:\Temp\File.txt* file, the `StreamReader` constructor throws a `FileNotFoundException`. In Example 8-1 that caused our program to crash, but because Example 8-3 has a catch block for that exception, the CLR will run that catch block. At this point, it will consider the exception to have been handled, so the program does not crash. Our catch block is free to do whatever it wants, and in this case my code just displays a message indicating that it couldn't find the file.

Exception handlers do not need to be in the method in which the exception originated. The CLR walks up the stack until it finds a suitable handler. If the failing `StreamReader` constructor call were in some other method that was called from inside the try block in Example 8-3, our catch block would still run (unless that method provided its own handler for the same exception).

Exception Objects

Exceptions are objects, and their type derives from the `Exception` base class.[1] This defines properties providing information about the exception, and some exception types have additional properties specific to the problem. Your catch block can get a reference to the exception if it needs information about what went wrong. Example 8-4 shows a modification to the catch block from Example 8-3. In the parentheses after the `catch` keyword, as well as specifying the exception type, we also provide an identifier name (`x`) with which we can refer to the exception object. This enables the code to read a property specific to the `FileNotFoundException` class: `FileName`.

Example 8-4. Using the exception in a catch block

```
try
{
```

[1] Strictly speaking, the CLR allows any type as an exception. However, C# can only throw `Exception`-derived types. Some languages let you throw other types, but it is strongly discouraged. C# can handle exceptions of any type, but only because the compiler automatically sets a `RuntimeCompatibility` attribute on all components it produces, asking the CLR to wrap non-`Exception`-derived exceptions in a `RuntimeWrappedException`.

```
    ... same code as Example 8-3 ...
}
catch (FileNotFoundException x)
{
    Console.WriteLine("Couldn't find the file '{0}'", x.FileName);
}
```

This will print out the name of the file that could not be found. With this simple program we already knew which file we were trying to open, but you could imagine this property being helpful in a more complex program that dealt with multiple files.

The general purpose members defined by the base `Exception` class include the `Message` property, which returns a string containing a textual description of the problem. The default error handling for console applications displays this text; the text "Could not find file 'C:\Temp\File.txt'." that we saw when first running Example 8-1 came from the `Message` property. This property is important when diagnosing unexpected exceptions.

`Exception` also defines an `InnerException` property. This is often null, but it comes into play when one operation fails as a result of some other failure. Sometimes, exceptions that occur deep inside a library would make little sense if they were allowed to propagate all the way up to the caller. For example, .NET provides a library for parsing XAML files. (XAML is a markup language used by various .NET user interface frameworks. I'll describe it in Chapter 21.) XAML is extensible, so it's possible that your code (or perhaps some 3rd party code) will run as part of the process of loading a XAML file, and this extension code could fail—suppose a bug in your code causes an `IndexOutOfRangeException` to thrown while trying to access an array element. It would be somewhat mystifying for that exception to emerge out of Microsoft's API, so regardless of the underlying cause of the failure, the library throws a `XamlParseException`. This means that if you want to handle the failure to load a XAML file, you know exactly which exception to handle, but the underlying cause of the failure is not lost: when some other exception caused the failure, it will be in the `InnerException`.

All exceptions contain information about where the exception was thrown. The `StackTrace` property provides the call stack as a string. As you've already seen, the default exception handler for console applications prints that. There's also a `TargetSite` property, which tells you which method was executing. It returns an instance of the reflection API's `MethodBase` class. See Chapter 13 for details on reflection.

Multiple Catch Blocks

A `try` block can be followed by multiple `catch` blocks. If the first catch does not match the exception being thrown, the CLR will look at the next one, then the next, and so on. Example 8-5 supplies handlers for both `FileNotFoundException` and `IOException`.

Example 8-5. Handling multiple exception types

```
try
{
    using (StreamReader r = new StreamReader(@"C:\Temp\File.txt"))
    {
```

```
        while (!r.EndOfStream)
        {
            Console.WriteLine(r.ReadLine());
        }
    }
}
catch (FileNotFoundException x)
{
    Console.WriteLine("Couldn't find the file '{0}'", x.FileName);
}
catch (IOException x)
{
    Console.WriteLine("IO error: '{0}'", x.Message);
}
```

An interesting feature of this example is that `FileNotFoundException` derives from `IOException`. I could remove the first catch block, and this would still handle the exception correctly, just with a less specific message, because the CLR considers a catch block to be a match if it handles the base type of the exception. So Example 8-5 has two viable handlers for a `FileNotFoundException`, and in these cases, C# requires the more specific one to come first. If I were to swap them over so that the `IOException` handler came first, I'd get this compiler error for the `FileNotFoundException` handler:

```
error CS0160: A previous catch clause already catches all exceptions of this or
of a super type ('System.IO.IOException')
```

If you write a catch block for the `Exception` base type, that will catch all exceptions. In most cases, this is the wrong thing to do—unless there is some specific and useful thing you can do with an exception, you should normally let it pass. Otherwise, you risk masking a problem. If you let the exception carry on, it's more likely to get to a place where it will be noticed, increasing the chances that you will fix the problem properly at some point. The one case in which a catch-all exception handler might make sense is if it's at a point where the only place left for the exception to go would be the default handling supplied by the system. (That might mean the `Main` method for a console application, but for multithreaded applications, it might mean the code at the top of a newly-created thread's stack.) It might be appropriate in these locations to catch all exceptions in order to be able to write the details to a log file or some similar diagnostic mechanism. Even then, once you've logged it you would probably want to rethrow the exception, as described later in this chapter.

> For critically important services, you might be tempted to write code that swallows the exception so that your application can limp on. This is not a good idea—if an exception you did not anticipate occurs, your application's internal state may no longer be trustworthy, because your code might have been half way through an operation when the failure occurred. If you cannot afford for the application to go offline, the best approach is to arrange for it to restart automatically after a failure. A Windows Service can be configured to do this automatically, and IIS has a similar feature. Third party products are also available if these mechanisms do not suit your needs.

Nested Try Blocks

If an exception occurs in a try block that does not provide a suitable handler, the CLR will keep looking. It will walk up the stack if necessary, but you can have multiple sets of handlers in a single method by nesting one try/catch inside another try block, as Example 8-6 shows. `PrintFirstLineLength` nests a try/catch pair inside the try block of another try/catch pair. Nesting can also be done across methods—the `Main` method will catch any `NullReferenceException` that emerges from the `PrintFirstLineLength` method (which will be thrown if the file is completely empty—the call to `ReadLine` will return null in that case).

Example 8-6. Nested exception handling

```
static void Main(string[] args)
{
    try
    {
        PrintFirstLineLength(@"C:\Temp\File.txt");
    }
    catch (NullReferenceException)
    {
        Console.WriteLine("NullReferenceException");
    }
}

static void PrintFirstLineLength(string fileName)
{
    try
    {
        using (var r = new StreamReader(fileName))
        {
            try
            {
                Console.WriteLine(r.ReadLine().Length);
            }
            catch (IOException x)
            {
                Console.WriteLine("Error while reading file: {0}",
                    x.Message);
            }
        }
    }
    catch (FileNotFoundException x)
    {
        Console.WriteLine("Couldn't find the file '{0}'", x.FileName);
    }
}
```

The reason I nested the `IOException` handler here was to make it apply to one particular part of the work: it only handles errors that occur while reading the file, after it has been opened successfully. It might sometimes be useful to respond to that scenario differently than an error that prevented you from opening the file in the first place.

The cross-method handling here is somewhat contrived. The `NullReferenceException` could be avoided by testing the return value of `ReadLine` for null. However, the underlying CLR mechanism this illustrates is

extremely important. A particular try block can define catch blocks just for those exceptions it knows how to handle, letting the rest escape up to higher levels.

Letting exceptions carry on up the stack is often the right thing to do. Unless there is something useful your method can do in response to discovering an error, it's going to need to let its caller know there's a problem, so unless you want to wrap the exception in a different kind of exception, you may as well let it through.

> If you're familiar with Java you may be wondering if C# has anything equivalent to checked exceptions. It does not. Methods do not formally declare the exceptions they throw, so there's no way the compiler can tell you if you have failed either to handle them or declare that your method might in turn throw them.

You can also nest a try block inside a catch block. This is important if there are ways in which your error handler can itself fail. For example, if your exception handler logs information about a failure to disk, that would fail if there's a problem with the disk.

It's possible to write a try block that never catches anything. It's illegal to write a try block that isn't followed directly by something, but that something doesn't have to be a catch block: it can be a *finally block*.

Finally Blocks

A finally block contains code that always runs once its associated try block has finished. It runs whether execution left the try block simply by reaching the end, or by returning from the middle, or by throwing an exception. The finally block will run even if you use a `goto` statement to jump right out of the block. Example 8-7 shows a finally block in use.

Example 8-7. A finally block

```
using Microsoft.OfficeInterop.PowerPoint;

[STAThread]
static void Main(string[] args)
{
    var pptApp = new Application();
    var pres = pptApp.Presentations.Open(args[0]);
    try
    {
        ProcessSlides(pres);
    }
    finally
    {
        pres.Close();
    }
}
```

This is an excerpt from a utility I wrote to process the contents of a Microsoft Office PowerPoint file. This just shows the outermost code—I've omitted the actual detailed processing code because it's not relevant here (although if you're curious, it exports animated slides as video clips). I'm showing it because it uses `finally`. This example uses COM interop (which I'll describe in detail in Chapter 23) to control the PowerPoint application. This example closes the file once it has finished, and the reason I put that

code in a finally block is that I don't the program to leave things open if something goes wrong part way through. It's important because of the way COM automation works. It's not like opening a file, where the operating system automatically closes everything when the process terminates. If this program exits suddenly, PowerPoint will not close whatever had been opened—it just assumes that you meant to leave things open. I don't want that, and closing the file in a finally block is a reliable way to avoid it.

Normally you'd write a `using` statement for this sort of thing, but PowerPoint's COM-based automation API doesn't support .NET's `IDisposable` interface. In fact as we saw in the previous chapter, the `using` statement works in terms of finally blocks under the covers, as does `foreach`, so you're relying on the exception handling system's finally mechanism even when you write `using` statements and `foreach` loops.

> Finally blocks run correctly when your exception blocks are nested. If some method throws an exception which is handled by a method that's, say, five levels above it in the call stack, and if some of the methods in between were in the middle of `using` statements, `foreach` loops, or try blocks with associated finally blocks, all of these intermediate finally blocks (whether explicit, or generated implicitly by the compiler) will execute before the handler runs.

Handling exceptions is only half of the story, of course. Your code may well detect problems, and exceptions may be an appropriate mechanism for reporting them.

Throwing Exceptions

Throwing an exception is very straightforward. You simply construct an exception object of the appropriate type, and then use the `throw` keyword. Example 8-8 does this when it is passed a null argument.

Example 8-8. Throwing an exception

```
public static int CountCommas(string text)
{
    if (text == null)
    {
        throw new ArgumentNullException("text");
    }
    return text.Count(ch => ch == ',');
}
```

The CLR does all of the work for us. It captures the information required for the exception to be able to report its location through properties like `StackTrace` and `TargetSite`. (It doesn't calculate their final values, because these are relatively expensive to produce. It just makes sure that it has the information it needs to be able to produce them if asked.) It then hunts for a suitable try/catch block, and if any finally blocks need to be run, it'll execute those.

Rethrowing Exceptions

Sometimes it is useful to write a catch block that performs some work in response to an error, but which allows the error to continue once that work is complete. There's an obvious, but wrong way to do this, illustrated in Example 8-9.

Example 8-9. How NOT to rethrow an exception

```
try
{
    DoSomething();
}
catch (IOException x)
{
    LogIOError(x);
    // This next line is BAD!
    throw x;  // Do not do this
}
```

This will compile without errors, and it will even appear to work, but it has a serious problem: it loses the context in which the exception was originally thrown. The CLR treats this as a brand new exception and will reset the location information. The `StackTrace` and `TargetSite` will report that the error originated inside your catch block. This could make it hard to diagnose the problem, because you won't be able to see where it was originally thrown. Example 8-10 shows how you can avoid this problem.

Example 8-10. Rethrowing without loss of context

```
try
{
    DoSomething();
}
catch (IOException x)
{
    LogIOError(x);
    throw;
}
```

The only difference (aside from removing the warning comments) is that I'm using the `throw` keyword without specifying which object to use as the exception. You're only allowed to do this inside a catch block, and it rethrows whichever exception the catch block was in the process of handling. This means that the `Exception` properties that report the location from which the exception was thrown will still refer to the original throw location, and not the rethrow.

The feature built into Windows known as Windows Error Reporting (WER) complicates things slightly.[2] This is the component that leaps into action when an application crashes, and which, depending on how your machine is configured, can offer options including restarting the application, reporting the crash to Microsoft, debugging it, or just terminating it. In addition to all that, when a Windows application crashes, WER captures several pieces of information to identify the crash location. For .NET applications, this includes the name, version, and timestamp of the component that failed, the exception type that was thrown, and it identifies not just the method, but also the offset into that method's IL from which the exception was thrown. These pieces of information are sometimes referred to as the *bucket* values. If the application crashes twice with the same values, those two crashes go into the same bucket, meaning that they are considered to be in some sense the same crash.

[2] Some people refer to WER by the name of an older Windows crash reporting mechanism: Dr. Watson. Some reduce this further to just Watson, or more cryptically, "a house call from the Dr."

Crash bucket values are not exposed as public properties of exceptions, but you can see them in the Windows event log. In the Event Viewer application, these log entries show up in the "Application" section under "Windows Logs" and the Source and Event ID columns for these entries will contain "Windows Error Reporting" and 1001 respectively. WER reports various kinds of crashes, so if you open a WER log entry, it will contain an Event Name value. For .NET crashes, this will be `CLR20r3`. The assembly name and version are easy enough to spot, as is the exception type. The method is more obscure: it's on the line labeled P7, but it's just a number based on the method's *metadata token*; if you want to find out what method that refers to, the ILDASM tool supplied with Visual Studio has a command line option to report the metadata tokens for all your methods.

Computers can be configured to upload crash reports to an error reporting service, and usually, just the bucket values get sent, although the services can request additional data. Bucket analysis can be useful when deciding how to prioritize bug fixes: it makes sense to start with the largest bucket because that's the crash your users are seeing most often. (Or at least, it's the one seen most often by users who have not disabled crash reporting. I always enable this on my computers, because I want the bugs I encounter in the programs I use to be fixed first.)

> The way to get access to accumulated crash bucket data depends on the kind of application you're writing. For a line-of-business application that only runs inside your enterprise, you will probably want to run an error reporting server of your own, but if the application runs outside of your administrative control, you can use Microsoft's own crash servers. There's a certificate-based process for verifying that you are entitled to the data, but once you've jumped through the relevant hoops, Microsoft will show you all reported crashes for your applications, sorted by bucket size.

Certain exception handling tactics can defeat the crash bucket system. If you write common error handling code that gets involved with all exceptions, there's a risk that WER will think that your application only ever crashes inside that common handler, which would mean that crashes of all kinds would go into the same bucket. This is not inevitable, but to avoid it, you need to understand how your exception handling code affects WER crash bucket data.

If an exception rises to the top of the stack without being handled, WER will get an accurate picture of exactly where the crash happened, but things may go wrong if you catch an exception before eventually allowing it (or some other exception) to continue up the stack. The behavior depends on which version of .NET you use. Before .NET 4.0, rethrowing an exception would only preserve the original location for the WER bucket values if you used the approach in Example 8-10, and not with the bad approach shown in Example 8-9. Slightly surprisingly, .NET 4.0 and .NET 4.5 preserves the location for WER in both cases. (From a .NET perspective, Example 8-9 loses the exception context for all versions—the `StackTrace` will show the rethrow location. Example 8-10 will preserve this.) It's a similar story when you wrap an exception as the `InnerException` of a new one—before .NET 4.0, WER would use the site of the outer exception for the bucket values, but with 4.0 and 4.5, if the exception that crashes an application had a non-null `InnerException`, that inner exception's location is used for the crash bucket.

This means that in .NET 4.0 or later, it's relatively easy to preserve the WER bucket. The only ways to lose the original context are either to handle the exception completely (i.e., not to crash) or to write a catch block that handles the exception and then throws a new one without passing the original one in as an `InnerException`. But if for some reason you have to use an older version of .NET, you need to be more careful—a bad rethrow of the kind shown in Example 8-9 will lose the context for both .NET and WER; throwing a new exception while wrapping the original as the `InnerException` will keep the full call stack available from a .NET perspective, but WER will only see the location at which the outer exception was thrown.

> The behavior I've just described for pre-.NET 4.0 is based on how those versions work on a system with all available service packs and updates installed at the time of writing this. Some web sites and books contradict this, claiming that even Example 8-10 would prevent WER from recording the original location of the underlying fault. This may have been true with the original release of .NET 2.0, does not appear to be true with current service packs applied. So be aware that these sorts of details can change from time to time.

Although Example 8-10 preserves the original context, this approach has a limitation: you can only rethrow the exception from inside the block in which you caught it. As asynchronous programming becomes more prevalent, it will become more common for exceptions to occur on some random worker thread. We need a reliable way to capture the full context of an exception, and to be able to rethrow it with that full context some arbitrary amount of time later, possibly from a different thread.

.NET 4.5 introduces a new class that solves these problems: `ExceptionDispatchInfo`. If you call its static `Capture` method from a catch block, passing in the current exception, it captures the full context, including the information required by WER. The `Capture` method returns an instance of `ExceptionDispatchInfo`. When you're ready to rethrow the exception, you can call this object's `Throw` method, and the CLR will rethrow the exception with the original context fully intact. Unlike the mechanism shown in Example 8-10, you don't need to be inside a catch block when you rethrow. You don't even need to be on the thread from which the exception was originally thrown.

Failing Fast

Some situations call for drastic action. If you detect that your application is in a hopelessly corrupt state, throwing an exception may not be sufficient, because there's always the chance that something may handle it and then attempt to continue. This risks corrupting persistent state—perhaps the invalid in-memory state could lead to your program writing bad data into a database. It may be better to bail out immediately before you do any lasting damage.

The `Environment` class provides a `FailFast` method. If you call this, the CLR will write a message to the Windows event log and will then terminate your application, providing details to the Windows Error Reporting service if that has been enabled on the computer. You can pass a string to be included in the event log entry, and you can also pass an exception, in which case the exception's details will also be written to the log, including the WER bucket values for the point at which the exception was thrown.

Exception Types

When your code detects a problem and throws an exception, you need to choose which type of exception to throw. You can define your own exception types, but the .NET Framework class library defines a large number of exception types, so in a lot of situations, you can just pick an existing type. There are hundreds of exception types, so a full list would be inappropriate here—if you want to see the complete set, the online documentation for the `Exception` class lists the derived types. However, there are certain ones that it's important to know about.

The class library defines an `ArgumentException` class, which is the base of several exceptions that indicate that a method has been called with bad arguments. Example 8-8 used `ArgumentNullException`, and there's also `ArgumentOutOfRangeException`. The base `ArgumentException` defines a `ParamName` property which contains the name of the parameter that was supplied with a bad argument. This is important for multi-argument methods because the caller will need to know which one was wrong. All these exception types have constructors that let you specify the parameter name, and you can see one of these in use in Example 8-8. The base `ArgumentException` is a concrete class, so if the argument is wrong in a way that is not covered by one of the derived types, you can just throw the base exception, providing a textual description the problem.

Besides the general purpose types just described, some APIs derive more specialized argument exceptions. For example, the `System.Globalization` namespace defines an exception type called `CultureNotFoundException` that derives from `ArgumentException`. You can do something similar, and there are two reasons for doing this. If there is additional information you can supply about why the argument is invalid, you will need a custom exception type so you can attach that information to the exception. (`CultureNotFoundException` provides three properties describing aspects of the culture information for which it was searching.) Alternatively, it might be that a particular form of argument error could be handled specially by a caller. Often, an argument exception simply indicates a programming error, but in situations where it might indicate an environment or configuration problem (e.g., not having the right language packs installed), developers might want to handle that specific issue differently. Using the base `ArgumentException` would be unhelpful in that case, because it would be hard to distinguish between the particular failure they want to handle, and any other problem with the arguments.

Some methods may want to perform work that could produce multiple errors. Perhaps you're running some sort of batch job, and if some individual tasks in the batch fail, you'd like to abort those, but to carry on with the rest, reporting all the failures at the end. For these scenarios, it's worth knowing about `AggregateException`. This extends the `InnerException` concept of the base `Exception`, adding an `InnerExceptions` property that returns a collection of exceptions.

Another commonly used type is `InvalidOperationException`. You would throw this if someone tries to do something with your object that it cannot support in its current state. For example, suppose you have written a class that represents a request that can be sent to a server. You might design this in such a way that each instance can only be used once, so if the request has already been sent, trying to modify the request further would be a mistake, and this would be an appropriate exception to throw. Another important

example is if your type implements `IDisposable`, and someone tries to use an instance after it has been disposed. That's a sufficiently common case that there's a specialized type derived from `InvalidOperationException` called `ObjectDisposedException`.

You should be aware of the distinction between `NotImplementedException` and the similar-sounding but semantically different `NotSupportedException`. The latter should be thrown when an interface demands it. For example, the `IList<T>` interface defines methods for modifying collections, but does not require collections to be modifiable—instead it says that read-only collections should throw `NotSupportedException` from members that would modify the collection. An implementation of `IList<T>` can throw this, and still be considered to be complete, whereas `NotImplementedException` means something is missing. You will most often see this in code generated by Visual Studio. The IDE can generate stub methods if you ask it to generate an interface implementation, or provide an event handler. It generates this code to save you from having to type in the full method declaration, but it's still your job to implement the body of the method, so Visual Studio will often supply a method that throws this exception so that you do not accidentally leave an empty method in place.

You would normally want to remove all code that throws `NotImplementedException` before shipping, replacing it with appropriate implementations. However, there is a situation in which you might want to leave it in place. Suppose you've written a library containing an abstract base class, and your customers write classes that derive from this. When you release new versions of the library, you can add new methods to that base class. Now imagine that you want to add a new library feature for which it would seem to make sense to add a new abstract method to your base class. That would be a breaking change—existing code that successfully derives from the old version of the class would no longer work. You can avoid this problem by providing a virtual method instead of an abstract method, but what if there's no useful default implementation that you can provide? In that case you might write a base implementation that throws a `NotImplementedException`. Code built against the old version of the library will not attempt to use the new feature, so it would never even attempt to invoke the method. But if a customer tried to use the new library feature without overriding the relevant method in their class, they would then get this exception. In other words, this provides a way to enforce a requirement of the form: you must override this method if and only if you want to use the feature it represents.

There are of course other, more specialized exceptions built in, and you should always try to find an exception that matches the problem you wish to report. However, you will sometimes need to report an error for which the class library does not supply a suitable exception. In this case, you will need to write your own exception class.

Custom Exceptions

The minimum requirement for a custom exception type is that it should ultimately derive from `Exception`. However, there are some design guidelines. The first thing to consider is the base class, and if you look at the built-in exception types, you'll notice that many of them derive only indirectly from `Exception`, through either `ApplicationException` or `SystemException`. You should avoid both of these. They were originally introduced with the intention of distinguishing between exceptions

produced by applications and ones produced by the system. However, this did not prove to be a useful distinction. Some exceptions could be thrown by both in different scenarios, and in any case, it was not normally useful to write a handler that caught all application exceptions but not all system ones, or vice versa. The class library design guidelines now tell you to avoid these two base types.

Custom exception classes normally derive directly from `Exception`, unless they represent a specialized form of some existing exception. For example, we already saw that `ObjectDisposedException` is a special case of `InvalidOperationException`, and the class library defines several more specialized derivatives of that same base class, such as `ProtocolViolationException` for networking code. If the problem you wish your code to report is clearly an example of some existing exception type, but it still seems useful to define a more specialized type, then you should derive from that existing type.

Although the `Exception` base class has a parameterless constructor, you should not normally use it. Exceptions should provide a useful textual description of the error, so your custom exception's constructors should all call one of the `Exception` constructors that take a string. You can either hard-code the message string[3] in your derived class, or define a constructor that accepts a message, passing it on to the base class; it's common for exception types to provide both, although that might be a waste of effort if your code only uses one of the constructors. It depends on whether your exception might be thrown by other code, or just yours.

It's also common to provide a constructor that accepts another exception, which will become the `InnerException` property value. Again, if you're writing an exception entirely for your own code's use, there's not much point in adding this constructor until you need it, but if your exception is part of a library, this is a common feature. Example 8-11 shows a hypothetical example that offers various constructors, along with an enumeration type which is used by the property the exception adds.

Example 8-11. A custom exception

```
public class DeviceNotReadyException : InvalidOperationException
{
    public DeviceNotReadyException(DeviceStatus status)
        : this("Device must be in Ready state", status)
    {
    }

    public DeviceNotReadyException(string message, DeviceStatus status)
        : base(message)
    {
        Status = status;
    }

    public DeviceNotReadyException(string message, DeviceStatus status,
```

[3] You could also consider looking up a localized string with the facilities in the `System.Resources` namespace instead of hard-coding it. The exceptions in the .NET Framework class library all do this. It's not mandatory, because not all programs run in multiple regions, and even for those that do, exception messages will not necessarily be shown to end users.

```
                                    Exception innerException)
        : base(message, innerException)
    {
        Status = status;
    }

    public DeviceStatus Status { get; private set; }
}

public enum DeviceStatus
{
    Disconnected,
    Initializing,
    Failed,
    Ready
}
```

The justification for a custom exception here is that this particular error has something more to tell us besides the fact that something was not in a suitable state. It provides information about the object's state at the moment at which the operation failed.

Although Example 8-11 is representative of typical custom exception types, it is technically missing something. If you look at the base `Exception` type, you'll see that it implements `ISerializable`, and is marked with the `[Serializable]` attribute. This is a special attribute recognized by the runtime: it gives the CLR permission to convert the object into a byte stream, which can later be converted back into an object, perhaps in a different process, and maybe even on a different machine. The runtime can automate these conversions entirely, but the `ISerializable` interface allows objects to customize the process.

The .NET Framework class library design guidelines recommend that exceptions should be serializable. This enables them to cross between *appdomains*. An appdomain is an isolated execution context. Programs that run in separate processes are always in separate appdomains, but it's possible to divide a single process into multiple appdomains. A fatal crash that terminates one appdomain need not bring down the entire process. Appdomains also provide a security boundary that prevents code in one appdomain from obtaining and using a direct reference to an object in another appdomain even if it's in the same process. Certain application hosting systems, such as the ASP.NET web framework can use appdomains to host multiple applications in a single process while keeping them isolated. By making an exception serializable, you make it possible for the exception to cross appdomain boundaries—the object cannot be used directly across the boundary, but serialization enables a copy of the exception to be built in the target appdomain. This means an exception thrown by a hosted application can be caught and logged by the host even if the host had pushed the application into its own separate appdomain.

If you don't need to support this scenario, you don't need to make your exceptions serializable, but for completeness, I'll just describe the changes you would need to make. First, serialization support is not inherited—just because your base class is serializable, that doesn't automatically mean your class is. So you would need to add the `[Serializable]` attribute in front of the class declaration. Then, because `Exception` opts into custom serialization, we have to follow suit, which means overriding the one and only member of `ISerializable`, but also providing a special constructor that the runtime will use when deserializing your type. Example 8-12 shows the members you would need to add to make the custom exception in Example 8-11

support serialization. The `GetObjectData` method simply stores the current value of the exception's `Status` property in a name/value container that the CLR supplies during serialization. It retrieves this value in the constructor that gets called during deserialization.

Example 8-12. Adding serialization support

```
public override void GetObjectData(SerializationInfo info,
                                   StreamingContext context)
{
    base.GetObjectData(info, context);
    info.AddValue("Status", Status);
}

public DeviceNotReadyException(SerializationInfo info,
                               StreamingContext context)
    : base(info, context)
{
    Status = (DeviceStatus) info.GetValue("Status", typeof(DeviceStatus));
}
```

Another feature to consider with a custom exception is whether to set the base `Exception` class's `HResult` property. This is a protected property, so it's not something a .NET exception handler would ever use, but it becomes significant if your exception reaches an interop boundary. (.NET interop services are described in Chapter 23.) If your .NET code is called through an interop mechanism, a .NET exception cannot propagate out into unmanaged code. Instead, the `HResult` property will determine the error code that unmanaged callers see. The property should therefore return the COM error code that is the nearest equivalent to the error that the exception represents. Not all .NET exceptions will have corresponding error codes. Some of the built in ones do: `FileNotFoundException` sets `HResult` to 0x80070002 for example. If you're familiar with COM errors (which have the type `HRESULT` in the Win32 SDK) you'll know that the 0x8007 prefix indicates that this is actually a Win32 error code wrapped as an `HRESULT`, so this is the COM equivalent of the Win32 `ERROR_FILE_NOT_FOUND` error code.

The base class will provide a value, so you don't have to set this. If you derive directly from `Exception`, `HResult` will be 0x80131500. (0x8013 is the COM error prefix for .NET errors.) Example 8-11 derives from `InvalidOperationException`, which sets its `HResult` to 0x80131509. As it happens, there is a better Win32 equivalent for the particular problem our exception represents: ERROR_NOT_READY, which has the value 0x15, so the `HRESULT` equivalent would be 0x80070015. If there's any chance that the exception might make it to an interop boundary at which it would need to be interpreted correctly, then we should set the `HResult` property to that value in the exception's constructors.

Unhandled Exceptions

Earlier, you saw the default behavior that a console application exhibits when your code throws an exception that it does not handle. It displays the exception's type, message, and stack trace and then terminates the process. This happens whether the exception went unhandled on the main thread or a thread you created explicitly, or even a thread pool thread that the CLR created for you. (This was not always true. Before .NET 2.0, threads

created for you by the CLR would implicitly swallow exceptions without either reporting them or crashing. You may occasionally encounter applications that still work this way: if the application configuration file contains a `legacyUnhandledExceptionPolicy` element with an `enabled="1"` attribute, the old .NET v1 behavior returns, meaning that unhandled exceptions can vanish silently.)

The CLR provides a way to discover when unhandled exceptions reach the top of the stack. The `AppDomain` class provides an `UnhandledException` event which the CLR raises when this happens on any thread. I'll be describing events in Chapter 9, but jumping ahead a little, Example 8-13 shows how to handle this event, and also throws an unhandled exception to try the handler out.

Example 8-13. Unhandled exception notifications

```
static void Main(string[] args)
{
    AppDomain.CurrentDomain.UnhandledException += OnUnhandledException;

    // Crash deliberately to illustrate UnhandledException event
    throw new InvalidOperationException();
}

private static void OnUnhandledException(object sender,
    UnhandledExceptionEventArgs e)
{
    Console.WriteLine("An exception went unhandled: {0}", e.ExceptionObject);
}
```

When the handler is notified, it's too late to stop the exception—the CLR will terminate the process shortly after calling your handler. The main reason this event exists is to provide a place to put logging code so that you can record some information about the failure for diagnostic purposes. In principle, you could also attempt to store any unsaved data to facilitate recovery if the program restarts, but you should be careful: if your unhandled exception handler gets called, then by definition your program is in a suspect state, so whatever data you save may be invalid.

Some application frameworks provide their own ways to deal with unhandled exceptions. For example, desktop applications for Windows need to run a message loop to respond to user input and system messages. This is typically supplied by some UI framework (e.g., Windows Forms or WPF). The framework's message loop inspects each message and may decide to call one or more methods in your code, and it will usually wrap each call in a try block, so that it can catch any exceptions your code may throw. One reason for this is that the default behavior of printing out details to the console is not very useful for applications that don't show a console window. The frameworks may show error information in a window instead. And web frameworks such as ASP.NET, need a different mechanism: at a minimum they should generate a response that indicates a server-side error in the way recommended by the HTTP specification.

This means that the `UnhandledException` event that Example 8-13 uses may not be raised when an unhandled exception escapes from your code, because it may be caught by a framework. If you are using an application framework, you should check to see if it provides its own mechanism for dealing with unhandled exceptions. For example, ASP.NET applications can have a *global.asax* file with various global event handlers, and if this contains an `Application_Error` method, you can deal with unhandled exceptions in there. WPF has its own `Application` class, and its

`DispatcherUnhandledException` event is the one to use. Likewise, Windows Forms provides an `Application` class with a `ThreadException` member.

Even when using these frameworks, their unhandled exception mechanisms only deal with exceptions that occur on threads the frameworks control. If you create a new thread and throw an unhandled exception on that, it would show up in the `AppDomain` class's `UnhandledException` event, because frameworks don't control the whole CLR.

Debugging and Exceptions

By default, Visual Studio's debugger will step in if an unhandled exception occurs while it is attached, but if the CLR is able to find a handler for an exception, the debugger will allow that to run without interruption. This can be a problem in situations where frameworks perform their own unhandled exception management—from the CLR's perspective, an exception may appear to have been handled because some UI framework's message loop had a try/catch in place when it called your handler. To some extent, frameworks can mitigate this by collaborating with the debugger—if you write a click event handler for a button in a WPF application, and you throw an exception from that handler, the debugger will in fact step in because WPF is in cahoots with Visual Studio. However, in more complicated scenarios, it's possible that by the time the debugger decides to step in, you are some way away from the original exception, because it has been wrapped in some other exception.

For example, if you write a particular kind of reusable WPF user interface component called a *user control*, and if it throws an exception in its constructor, the debugger will not necessarily break in at the point at which that exception is thrown. If you use your user control from within XAML, the XAML parser will catch the exception and will, as mentioned earlier, wrap it as the `InnerException` of a `XamlParseException`. The debugger will typically break in only when that wrapper exception is thrown, and not when your code threw the original exception. You'll be able to find out where the original error occurred by inspecting the `InnerException`, but you won't be able to look at the local variables or any other state that was in place at the point at which the problem occurred, because the thread has moved on.

For this reason, I frequently reconfigure Visual Studio so that it breaks in as soon as exceptions are thrown, even if a handler is available. This means that the debugger can show you the full context in which the exception occurred. You can set this behavior with the Exceptions dialog, shown in Figure 8-2. This is available from the Debug menu.

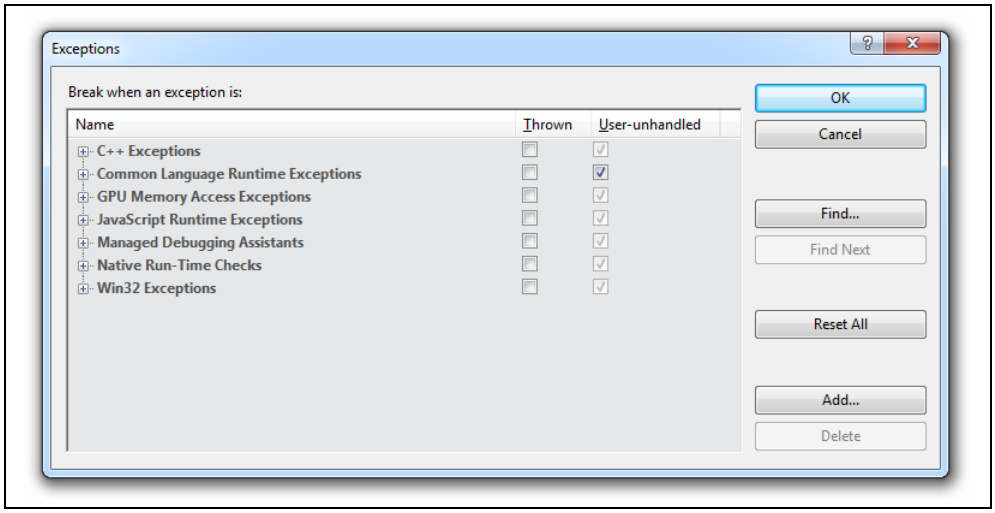

Figure 8-2. Visual Studio's Exceptions dialog

If you check the box in the *Thrown* column, the debugger will break in any time any exception is thrown. (This dialog handles all different kinds of code. For .NET applications, you'd check the box on the *Common Language Runtime Exceptions* line.) If the Thrown box is not checked, the *User-unhandled* column lets you choose whether to break in when the exception is handled by code you didn't write (e.g., in a catch block supplied a class library component), or only to break in when the exception is entirely unhandled. You won't always see that second column by the way—it depends on Visual Studio being able to make a distinction between your code and other code, which it can only do if the "Just My Code" feature has been enabled in the debugging page of the Options dialog. That option is incompatible with some features, including the one that automatically downloads the source code of the .NET Framework class libraries so that you can step through that as well (also configured with the Options dialog). But the *Thrown* column will always be present.

One problem with debugging exceptions as soon as they are thrown is that some code throws a lot of benign exceptions. Some frameworks seem to do this more than others—ASP.NET seems to throw and then immediately catch a few inconsequential exceptions as a matter of course during startup, whereas WPF rarely throws exceptions unless something is wrong. So depending on the sort of application you're writing, you may need to be more selective. If you expand the *Common Language Runtime Exceptions* node, it shows a treeview of exception types broken down by namespace, so you can configure different behavior for different exceptions. You can add custom exception types to this dialog with its *Add* button to customize behavior for your own exception types. Unfortunately, there's no way to configure location-specific behavior—so if you know a particular application or framework will always throw and catch an exception in a particular place and you'd always like to ignore that, but you want to see the same exception type any other place it gets thrown, you can't.

Asynchronous Exceptions

Back at the start of this chapter, I mentioned that the CLR can throw certain exceptions at any point during your code's execution, and that these exceptions may be caused by

factors outside of your control. These are called asynchronous exceptions, although they have nothing to do with asynchronous programming or the `async` keyword described in Chapter 18. In this context, 'asynchronous' merely means that the events that cause these exceptions can happen independently of what your code may be doing at the time.

The exceptions that can occur asynchronously are `ThreadAbortException`, `OutOfMemoryException`, and `StackOverflowException`. The first of these can occur if some other thread decides to abort yours. The CLR will do that if necessary when shutting down an appdomain, but you can also do it programmatically by calling the relevant `Thread` object's `Abort` method. The other two are more surprising—you might not expect these to be able to emerge at any point in the code. Surely you'd only run out of memory while attempting to allocate memory? And surely you'd only see a stack overflow if attempting some operation that needs more stack such as a function call? Well, the CLR reserves the right to grow the stack dynamically in the middle of a method to make space for temporary storage, and it also reserves the right to perform other memory allocations at any time for any reason it sees fit. That's why these other two exceptions are considered asynchronous—your code could indirectly cause heap or stack use at any time, even if you did not explicitly ask for it.

Asynchronous exceptions present a challenge when it comes to cleaning up resources because they can occur inside finalizers and finally blocks (including implicit ones such as those generated by a `using` statement). If your code calls into unmanaged code and obtains handles, how can it guarantee to free those handles in the face of asynchronous exceptions? Even if you've carefully written using statements, finally blocks, and finalizers to ensure that handles are freed in a timely fashion where possible, and eventually in any event, what can you do if an asynchronous exception occurs inside your finally block or finalizer just as you were about to close a handle?

You can solve this problem with a *constrained execution region* (CER). A CER is a block of code which the CLR guarantees will never encounter an asynchronous exception. The runtime can only offer this guarantee if your code avoids certain operations. You must not allocate memory explicitly with `new` or implicitly with a boxing operation. You must not attempt to acquire a lock for multithreading synchronization purposes. You cannot access a multidimensional array. Indirect method invocation is, in most cases, not allowed: a CER cannot use delegates or raw function pointers, you cannot invoke methods through the reflection API, and use of virtual methods is limited. (It's possible to invoke virtual methods, but you need to tell the CLR which particular implementations you plan to invoke before entering the CER.) In fact, use of other methods in general is limited—any method called by your CER is subject to the same limitations.

The purpose of all these constraints is to make it possible for the CLR to determine in advance whether it has enough memory to run the whole CER. It ensures that all of the code the CER will execute has already been JIT compiled. Any temporary storage on the heap or stack that the method could require will be allocated in advance, ruling out the possibility of an `OutOfMemoryException` or a `StackOverflowException` while the region runs. Thread aborts are blocked for the duration of the CER. (Of course, it won't necessarily prevent any of these exceptions, it just means that if they are going to occur, they will happen either before the CER begins to run, or after it has finished.)

There are three ways to write a CER. The first is to write a type that derives from `CriticalFinalizerObject` (directly, or indirectly, e.g. via `SafeHandle`) as discussed in Chapter 7. The finalizer of such a type is a CER, and the CLR won't allow the object to be created unless it is able to commit in advance to running the finalizer

eventually. The other two ways both involve the `RuntimeHelpers` class. This has a static `PrepareConstrainedRegions` method, and if you call this immediately before a `try` keyword, the CLR will treat all of that block's corresponding catch and finally blocks as CERs, and will commit to being able to execute any of them before starting to run the try block. (The try block itself will not be a CER.) The `RuntimeHelpers` class also provides an `ExecuteCodeWithGuaranteedCleanup` method which takes two delegates. The first delegate is executed normally, but the second is treated as a CER, and will be prepared before the first delegate is invoked, to guarantee that it can be run in any event.

Constrained execution regions are part of a broader set of CLR features sometimes referred to collectively as the *reliability* features. These are designed to ensure predictable behavior in the face of extreme scenarios such as running out of memory, or sudden appdomain termination. Writing code that is reliable in these situations is difficult, and the benefits may sometimes be doubtful—if your system has run out of memory, you may well have bigger problems at this point. These reliability features were added to make it possible for SQL Server to host the CLR, and to run unmanaged code without compromising the high availability standards people demand from their databases. Where possible, it's best to rely on code that uses these features for you, such as the various types that derive from `SafeHandle`. A full discussion of the use of these reliability features, and the specialized hosting environments for which they are designed such as SQL Server, is beyond the scope of this book.

Summary

In .NET, errors are usually reported with exceptions, apart from in certain scenarios where failure is expected to be common and the cost of exceptions is likely to be high compared to the cost of the work at hand. Exceptions allow error handling code to be separated out from code that does work. They also make it harder to ignore errors—unexpected errors will propagate up the stack and eventually cause the program to terminate and produce an error report. Catch blocks allow us to handle those exceptions that we can anticipate. (You can also use them to catch all exceptions indiscriminately, but that's usually a bad idea—if you don't know why a particular exception occurred, you cannot know for certain how to recover from it safely.) Finally blocks provide a way to perform cleanup safely regardless of whether code executes successfully or encounters exceptions. The .NET Framework class library defines numerous useful exception types, but if necessary we can write our own.

In the chapters so far, we've looked at the basic elements of code, classes and other custom types, collections, and error handling. There's one last feature of the C# type system to look at, and that's a special kind of object called a delegate.

9

Delegates, Lambdas, and Events

The most common way to use an API is to invoke the methods and properties its classes provide, but sometimes, things need to work in reverse. In Chapter 5, I showed the search features offered by arrays and lists. To use these, I wrote a method that returned true when its argument met my criteria, and the relevant APIs called my method for each item they inspected. Not all callbacks are immediate. Asynchronous APIs can call a method in our code when long-running work completes. In a client-side application, I want my code to run when the user interacts with certain visual elements in particular ways, such as clicking a button.

Interfaces and virtual methods can enable callbacks. In Chapter 4, I showed the `IComparer<T>` interface, which defines a single `CompareTo` method. This is called by methods like `Array.Sort` when we want a customized sort ordering. You could imagine a UI framework that defined an `IClickHandler` interface with a `Click` method, and perhaps also `DoubleClick`. The framework could require us to implement this interface if we want to be notified of button clicks.

In fact, none of .NET's UI frameworks use the interface-based approach, because it gets cumbersome when you need multiple kinds of callback. Single and double clicks are the tip of the iceberg for user interactions—in WPF application, each user interface element can provide over 100 kinds of notifications. Most of the time, you only need to handle one or two events from any particular element, so an interface with 100 methods to implement would be annoying.

Splitting notifications across multiple interfaces could mitigate this inconvenience. Also, a base class with virtual methods might help, because it could provide default, empty implementations for all callbacks, meaning we'd only need to override the ones we were interested in. But even with these improvements, there's a serious drawback with this object-oriented approach. Imagine a user interface with four buttons. In a hypothetical UI framework that used the approach I've just described, if you wanted different `Click` handler methods for each button, you'd need four distinct implementations of the `IClickHandler` interface. A single class can only implement any particular interface once, so you'd need to write four classes. That seems very cumbersome when all we really want to do is tell a button to call a particular method when clicked.

C# provides a much simpler solution in the form of a *delegate*, which is a reference to a method. If you want a library to call your code back for any reason, you will normally just pass a delegate referring to the method you'd like it to call. I showed an example of that in Chapter 5, which I've reproduced in Example 9-1. This finds the index of the first non-zero element in an `int[]` array.

Example 9-1. Searching an array using a delegate

```
public static int GetIndexOfFirstNonEmptyBin(int[] bins)
{
    return Array.FindIndex(bins, IsGreaterThanZero);
}

private static bool IsGreaterThanZero(int value)
{
    return value > 0;
}
```

At first glance, this seems very simple: the second parameter to `Array.FindIndex` requires a method that it can call to ask whether a particular element is a match, so I passed my `IsGreaterThanZero` method as an argument. But what does it really mean to pass a method, and how does this fit in with .NET's type system, the CTS?

Delegate Types

Example 9-2 shows the declaration of the `FindIndex` method used in Example 9-1. The first parameter is the array to be searched, but it's the second one we're interested in—that's where I passed a method.

Example 9-2. Method with a delegate parameter

```
public static int FindIndex<T>(
      T[] array,
      Predicate<T> match
  )
```

The method argument's type is `Predicate<T>`, where `T` is also the array element type. (Example 9-1 uses an `int[]`, so the second argument will be a `Predicate<int>`.) Example 9-3 shows how this type is defined. This is the whole of the definition, not simply an excerpt; if you wanted to write a type that was equivalent to `Predicate<T>`, that's all you'd need to write.

Example 9-3. The Predicate<T> delegate type

```
public delegate bool Predicate<in T>(T obj);
```

Breaking Example 9-3 down, we begin, like most type definitions, with the accessibility, and we can use all the same keywords we could for other types such as `public`, or `internal`. (Like any type, delegate types can be nested inside some other type, so they can also be `private` or `protected`.) Next is the `delegate` keyword, which just tells the C# compiler that we're defining a delegate type. The rest of the definition looks, not coincidentally, just like a method declaration. We have a return type of `bool`. You put the type name where you'd normally see the method name. The angle brackets indicate that this is a generic type with a single contravariant type argument `T`, and the method signature has a single parameter of that type.

Instances of delegate types are usually just called delegates, and they refer to methods. A method is compatible with (i.e., can be referred to by an instance of) a particular delegate type if its signature matches. The `IsGreaterThanZero` method in Example 9-1 takes an `int` and returns a `bool`, so it is compatible with `Predicate<int>`. The match does not have to be precise. If implicit reference conversions are available for parameter types, you can use a more general method. For example, a method with a return type of `bool`, and a single parameter of type `object`, would obviously be compatible with `Predicate<object>`, but because this method can accept `string` arguments, it would also be compatible with `Predicate<string>`. (It would not be compatible with `Predicate<int>` though because there's no implicit reference conversion from `int` to `object`. There's an implicit boxing conversion, but that's not the same thing.)

Creating a Delegate

You can use the `new` keyword to create a delegate. Where you'd normally pass constructor arguments, you can supply the name of a compatible method. Example 9-4 constructs a `Predicate<int>`, so it needs a method with a `bool` return type that takes an `int`. The `IsGreaterThanZero` method in Example 9-1 fits the bill. (You'd only be able to write this code where `IsGreaterThanZero` is in scope, i.e., inside the same class.)

Example 9-4. Constructing a delegate

```
var p = new Predicate<int>(IsGreaterThanZero);
```

In practice, we rarely use `new` for delegates. It's only necessary in cases where the compiler cannot infer the delegate type. Expressions that refer to methods are unusual in that they have no innate type—the expression `IsGreaterThanZero` is compatible with `Predicate<int>`, but there are other compatible delegate types. You could define your own non-generic delegate type that takes an `int` and returns a `bool`. Later in this chapter, I'll show the `Func` family of delegate types, and you could store a reference to `IsGreaterThanZero` in a `Func<int, bool>` delegate. So `IsGreaterThanZero` does not have a type of its own, which is why the compiler needs to know which particular delegate type we want. Example 9-4 assigns the delegate into a variable declared with `var`, which tells the compiler nothing about what type to use, which is why I've had to tell it explicitly with the constructor syntax.

In cases where the compiler knows what type is required, it can implicitly convert the method name to the target delegate type. In Example 9-5, the variable has an explicit type, so the compiler knows a `Predicate<int>` is required. This is equivalent to Example 9-4. Example 9-1 relies on the same mechanism—the compiler knows that the second argument to `FindIndex` is `Predicate<T>`, and because we supply a first argument of type `int[]`, it deduces that `T` is `int`, so it knows the second argument's full type is `Predicate<int>`. Having worked that out, it uses the same built-in implicit conversion rules to construct the delegate as Example 9-5.

Example 9-5. Implicit delegate construction

```
Predicate<int> p = IsGreaterThanZero;
```

When code refers to a method by name like this, the name is technically called a *method group*, because multiple overloads may exist for a single name. The compiler narrows this down by looking for the best possible match, in a similar way to how it chooses an

overload when you invoke a method. As with method invocation, it's possible that there will be either no matches or multiple equally good matches, in which case the compiler will produce an error.

Method groups can take several forms. In the examples shown so far, I have used an unqualified method name, which only works when the method in question is in scope. If you want to refer to a method defined in some other class, you would need to qualify it with the class name, as Example 9-6 shows.

Example 9-6. Delegates to methods in another class

```
internal class Program
{
    static void Main(string[] args)
    {
        Predicate<int> p1 = Comparisons.IsGreaterThanZero;
        Predicate<int> p2 = Comparisons.IsLessThanZero;
    }
}

internal class Comparisons
{
    public static bool IsGreaterThanZero(int value)
    {
        return value > 0;
    }

    public static bool IsLessThanZero(int value)
    {
        return value < 0;
    }
}
```

Delegates don't have to refer to static methods. They can refer to an instance method. There are a couple of ways you can make that happen. One is simply to refer to an instance method by name from a context in which that method is in scope. The `GetIsGreaterThanPredicate` method in Example 9-7 returns a delegate that refers to `IsGreaterThan`. Both are instance methods, so `IsGreaterThan` can only be used with an object reference, but `GetIsGreaterThanPredicate` has an implicit `this` reference, and the compiler automatically provides that to the delegate that it implicitly creates.

Example 9-7. Implicit instance delegate

```
public class ThresholdComparer
{
    public int Threshold { get; set; }

    public bool IsGreaterThan(int value)
    {
        return value > Threshold;
    }

    public Predicate<int> GetIsGreaterThanPredicate()
    {
        return IsGreaterThan;
    }
}
```

Alternatively, you can be explicit about which instance you want. Example 9-8 creates three instances of the `ThresholdComparer` class from Example 9-7, and then creates three delegates referring to the `IsGreaterThan` method, one for each instance.

Example 9-8. Explicit instance delegate

```
var zeroThreshold = new ThresholdComparer { Threshold = 0 };
var tenThreshold = new ThresholdComparer { Threshold = 10 };
var hundredThreshold = new ThresholdComparer { Threshold = 100 };

Predicate<int> greaterThanZero = zeroThreshold.IsGreaterThan;
Predicate<int> greaterThanTen = tenThreshold.IsGreaterThan;
Predicate<int> greaterThanOneHundred = hundredThreshold.IsGreaterThan;
```

You don't have to limit yourself to simple expressions of the form *`variableName.MethodName`*. You can take any expression that evaluates to an object reference, and then just append *`.MethodName`*, and if the object has one or more methods called *`MethodName`*, that will be a valid method group.

C# will not let you create a delegate that refers to an instance method without specifying either implicitly or explicitly which instance you mean, and it will always initialize the delegate with that instance.

> When you pass a delegate to some other code, that code does not need to know whether the delegate's target is a static or an instance method. And for instance methods it does not need to supply the instance. Delegates that refer to instance methods always know which instance they refer to, as well as which method.

There's another way to create a delegate that can be useful if you do not necessarily know which method or object you will use until runtime. The `Delegate` class has a static `CreateDelegate` method that lets you pass the delegate type, target object, and target method as arguments. There are a few ways of specifying the targets, so it has various overloads. They all take the delegate type's `Type` object as the first argument. (The `Type` class is part of the reflection API. I will explain it in detail, along with the `typeof` operator, in Chapter 13. As far as `CreateDelegate` is concerned, it's just a way to refer to a particular type.) Example 9-9 uses an overload which also takes the target instance and the name of the method.

Example 9-9. CreateDelegate

```
var greaterThanZero = (Predicate<int>) Delegate.CreateDelegate(
    typeof(Predicate<int>), zeroThreshold, "IsGreaterThan");
```

The other overloads include support for omitting the target object, which you would use for a static method, and for requesting case insensitivity for the method name. There are also overloads that accept the reflection API's `MethodInfo` object to identify the method instead of a string.

So a delegate combines two pieces of information: it identifies a specific function, and if that's an instance function, the delegate also contains an object reference. But some delegates do more.

Multicast Delegates

If you look at any delegate type with a reverse engineering tool such as ILDASM, you'll see that whether it's a type supplied by the .NET Framework class library, or one you've defined yourself, it derives from a base type called `MulticastDelegate`. As the name suggests, this means delegates can refer to more than one method. This is mostly only of interest in notification scenarios where you may need to invoke multiple methods when some event occurs. However, all delegates support this whether you need it or not.

Even delegates with non-`void` return types derive from `MulticastDelegate`. That doesn't usually make much sense. For example, code that requires a `Predicate<T>` will normally inspect the return value. `Array.FindIndex` uses it to find out whether an element matches our search criteria. If a single delegate refers to multiple methods, what's `FindIndex` supposed to do with multiple return values? As it happens, it will execute all the methods, but will ignore the return values of all except the final method that runs. (As you'll see in the next section, that's the default behavior you get if you don't provide any special handling for multicast delegates.)

The multicast feature is available through the `Delegate` class's static `Combine` method. This takes any two delegates and returns a single delegate. When the resulting delegate is invoked, it is as though you invoked the two original delegates one after the other. This works even when the arguments already refer to multiple methods—you can chain together ever larger multicast delegates. If the same method is referred to in both arguments, the resulting combined delegate will invoke it twice.

> Delegate combination always produces a new delegate. The `Combine` method does not modify the delegates you pass it.

In fact, we rarely call `Delegate.Combine` explicitly, because C# has built-in support for combining delegates. You can use the `+` or `+=` operators. Example 9-10 shows both, combining the three delegates from Example 9-8 into a single multicast delegate. The two resulting delegates are equivalent—this just shows two ways of writing the same thing. Both cases compile into a couple of calls to `Delegate.Combine`.

Example 9-10. Combining delegates

```
Predicate<int> megaPredicate1 =
    greaterThanZero + greaterThanTen + greaterThanOneHundred;

Predicate<int> megaPredicate2 = greaterThanZero;
megaPredicate2 += greaterThanTen;
megaPredicate2 += greaterThanOneHundred;
```

You can also use the `-` or `-=` operators, which produce a new delegate that is a copy of the first operand, but with its last reference to the method referred to by the second operand removed. As you might guess, this turns into a call to `Delegate.Remove`.

> Delegate removal behaves in a potentially surprising way if the delegate you remove refers to multiple methods. Subtraction of a multicast delegate only succeeds if the delegate from which you are subtracting contains all of the methods in the delegate being subtracted *sequentially and in the same order*. Given the delegates in Example 9-10, subtracting `(greaterThanTen +`

`greaterThanOneHundred)` from `megaPredicate1` would work, but subtracting `(greaterThanZero + greaterThanOneHundred)` would not, because although `megaPredicate1` contains references to the same two methods, and in the same order, the sequence is not exactly the same—`megaPredicate1` has an additional delegate in the middle.

Invoking a Delegate

So far I've shown how to create a delegate, but what if you're writing your own API that needs to call back into a method supplied by your caller? In other words, how do you consume a delegate? First, you would need to pick a delegate type. You could use one supplied by the class library, or if necessary, you can define your own. You can use this delegate type for a method parameter or a property. Example 9-11 shows what to do when you want to call the method (or methods) the delegate refers to.

Example 9-11. Invoking a delegate

```
public static void CallMeRightBack(Predicate<int> userCallback)
{
    bool result = userCallback(42);
    Console.WriteLine(result);
}
```

As this not terribly realistic example shows, you can use a variable of delegate type as though it were a function. Any expression that produces a delegate can be followed by an argument list in parentheses. The compiler will generate code that invokes the delegate. If the delegate has a non-`void` return type, the invocation expression's value will be whatever the underlying method returns (or in the case of a delegate referring to multiple methods, whatever the final method returns).

Delegates are special types in .NET, and they work quite differently than classes or structs. The compiler generates a superficially normal-looking class definition with various members that we'll look at shortly, but the members are all empty—C# produces no IL for any of them. The CLR provides the implementation at runtime. It does the work required to invoke the target method, including invoking all of the methods in multicast scenarios.

Although delegates are special types with runtime-generated code, there is ultimately nothing magical about invoking a delegate. The call happens on the same thread, and exceptions propagate through methods that were invoked via a delegate in exactly the same way as they would if the method were invoked directly. Invoking a delegate with a single target method works as though your code had called the target method in the conventional way. Invoking a multicast delegate is just like calling each of its target methods in turn.

If you want to get all the return values from a multicast delegate, you can take control of the invocation process. Example 9-12 retrieves an *invocation list* for a delegate, which is an array containing a single-method delegate for each of the methods to which the original multicast delegate refers. If the original delegate contained only a single method, this list will contain just that one delegate, but if the multicast feature is being exploited,

this provides a way to invoke each in turn. This enables the example to look at what each individual predicate says.

> Example 9-12 relies on a trick with `foreach`. The `GetInvocationList` method returns an array of type `Delegate[]`. The `foreach` loop nonetheless specifies an iteration variable type of `Predicate<int>`. This causes the compiler to generate a loop that casts each item to that type as it retrieves it from the collection.

Example 9-12. Invoking each delegate individually

```
public static void DemocracyInAction(Predicate<int> userCallbacks)
{
    int ayes = 0;
    int noes = 0;
    foreach (Predicate<int> p in userCallbacks.GetInvocationList())
    {
        bool result = p(42);
        if (result)
        {
            ayes += 1;
        }
        else
        {
            noes += 1;
        }
    }
    if (ayes > noes)
    {
        Console.WriteLine("The ayes have it");
    }
    else if (noes > ayes)
    {
        Console.WriteLine("The noes have it");
    }
    else
    {
        Console.WriteLine("It's a tie");
    }
}
```

There's one more way to invoke a delegate that is occasionally useful. The base `Delegate` class provides a `DynamicInvoke` method. You can call this on a delegate of any type without needing to know at compile time exactly what arguments are required. It takes a `params` array of type `object[]`, so you can pass any number of arguments. It will verify the number and type of arguments at runtime. This can enable certain late binding scenarios, although since C# 4 introduced intrinsic dynamic features (discussed in Chapter 14), it's more likely that you'd just use those in any new code.

Common Delegate Types

The .NET Framework class library provides several useful delegate types, and you will often be able to use these instead of needing to define your own. For example, it defines a set of generic delegates named `Action` with varying numbers of type parameters—

`Action<T>`, `Action<T1, T2>`, `Action<T1, T2, T3>` etc. These all follow a common pattern: for each type parameter, there's a single method parameters of that type. Example 9-13 shows the first four, including the zero-argument form.

Example 9-13. The first few Action delegates

```
public delegate void Action();
public delegate void Action<in T1>(T1 arg1);
public delegate void Action<in T1, in T2 >(T1 arg1, T2 arg2);
public delegate void Action<in T1, in T2, in T3>(T1 arg1, T2 arg2, T3 arg3);
```

Although this is clearly an open-ended concept—you could imagine delegates of this form with any number of arguments—the CTS doesn't provide a way to define this sort of type as a pattern, so the class library has to define each form as a separate type. Consequently, there's no 200-argument form of `Action`. The upper limit depends on the version of .NET. For the ordinary editions of .NET found on servers and desktops, version 3.5 only went as high as four arguments, but .NET 4 and 4.5 both go up to 16 arguments, as does the version of .NET available for Windows 8 Metro-style apps.

In Silverlight, which has its own release schedule and version numbering scheme, version 3 stopped at four arguments, but versions 4 and later also go up to 16 arguments.[1]

The one obvious limitation with `Action` is that these types have a `void` return type, so they cannot refer to methods that return values. But there's a similar family of delegate types, `Func`, that allow any return type. Example 9-14 shows the first few delegates in this family, and as you can see, they're pretty similar to `Action`. They just get an additional final type parameter, `TResult`, which specifies the return type.

Example 9-14. The first few Func delegates

```
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T1, out TResult>(T1 arg1);
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
public delegate TResult Func<in T1, in T2, in T3, out TResult>(
    T1 arg1, T2 arg2, T3 arg3);
```

Again, version 3.5 of the full CLR and version 3 of Silverlight support up to four arguments. Versions 4 and later of both go up to 16 arguments, as does the Windows 8 Metro version of .NET.

These two families of delegates would appear to have most requirements covered. Unless you're writing monster methods with more than 16 arguments, when would you ever need anything else? Why does the class library define a separate `Predicate<T>` when it could just use `Func<T, bool>` instead? In some cases the answer is history: many delegate types have been around since before these general-purposes types were added. But that's not the only reason—new delegate types continue to be added even now. The reason is that sometimes, it's useful to define a specialized delegate type to indicate particular semantics.

If you have a `Func<T, bool>`, all you know is that you've got a method that takes a `T` and returns a `bool`. But with a `Predicate<T>`, there's an implied meaning: it makes a

[1] The latest version of Windows phone at the time of writing this is v7.1, and it is based on Silverlight 3, so it also only goes up to four arguments.

decision about that T instance, and returns true or false accordingly; not all methods that take a single argument and return a bool necessarily fit that pattern. By providing a Predicate<T>, you're not just saying that you have a method with a particular signature, you're saying you have a method that serves a particular purpose. (As it happens, Predicate<T> was introduced before Func<T,bool>, so history is in fact the main reason why some APIs us it. However semantics still matter—there are some newer APIs for which Func<T,bool> was an option which nonetheless opted for Predicate<T>.)

The .NET Framework class library defines a huge number of delegate types, most of which are even more specialized than Predicate<T>. For example, the System.IO namespace and its descendants define several that relate to very specific events, such as SerialPinChangedEventHandler, which is only used when working with old-fashioned serial ports such as the once-ubiquitous RS232 interface.

Type Compatibility

Delegate types do not derive from one another. Any delegate type you define in C# will derive directly from MulticastDelegate, as do all of the delegate types in the class library. However, the type system supports certain implicit reference conversions for generic delegate types through covariance and contravariance. The rules are very similar to those for interfaces. As the in keyword in Example 9-3 showed, the type argument T in Predicate<T> is contravariant, which means that if an implicit reference conversion exists between two types, A and B, an implicit reference conversion also exists between the types Predicate<B> and Predicate<A>. Example 9-15 shows an implicit conversion that this enables.

Example 9-15. Delegate covariance

```
public static bool IsLongString(object o)
{
    var s = o as string;
    return s != null && s.Length > 20;
}


static void Main(string[] args)
{
    Predicate<object> po = IsLongString;
    Predicate<string> ps = po;
    Console.WriteLine(ps("Too short"));
}
```

The Main method first creates a Predicate<object> referring to the IsLongString method. Any target method for this predicate type is capable of inspecting any object of any kind, so it's clearly able to meet the needs of code that requires a predicate capable of inspecting strings, so it makes sense that the implicit conversion to Predicate<string> should succeed, which it does thanks to contravariance. Covariance also works in the same way as it does with interfaces, so it would typically be associated with a delegate's return type. (We denote covariant type parameters with the out keyword.) All of the built-in Func delegate types have a covariant type argument representing the function's return type called TResult. (The

type parameters for the function's parameters are all contravariant. This is also true for all of the type arguments for the `Action` delegate types.)

> The variance-based delegate conversions are implicit reference conversions. This means that when you convert the reference type, the result still refers to the same delegate instance. (Not all implicit conversions do this. Implicit numeric conversions create an instance of the target type; implicit boxing conversions create a new box on the heap.) So in Example 9-15, `po` and `ps` refer to the same delegate on the heap.

You might also expect delegates that look the same to be compatible. For example, a `Predicate<int>` can refer to any method that a `Func<int, bool>` can use, and vice versa, so you might expect an implicit conversion to exist between these two types. You might be further encouraged by the "Delegate compatibility" section in the C# specification which says that delegates with identical parameter lists and return types are compatible. (In fact it goes further, saying that certain differences are allowed. For example, I mentioned earlier that argument types may be different as long as certain implicit reference conversions are available.) However, if you try the code in Example 9-16, it won't work.

Example 9-16. Illegal delegate conversion

```
Predicate<string> pred = IsLongString;
Func<string, bool> f = pred;  // Will fail with compiler error
```

An explicit cast doesn't work either—if you manage to avoid the compiler error you'll just get a runtime error instead. The CTS considers these to be incompatible types, so a variable declared with one delegate type cannot hold a reference to a different delegate type even if their method signatures are compatible. This is not the scenario for which C#'s delegate compatibility rules are designed—they are mainly used to determine whether a particular method can be stored in a particular delegate.

The lack of type compatibility between 'compatible' delegate types may seem odd, but structurally identical delegate types don't necessarily have the same semantics. That's why some APIs choose a specialized delegate type such as `Predicate<T>` when a more general-purpose one would have worked. If you find yourself needing to perform this sort of conversion, it may be a sign that something is not quite right in your code's design.[2]

Having said that, it is possible to create a new delegate that refers to the same method as the original if the new type is compatible with the old type. It's always best to stop and ask why you find yourself needing to do that, but it's occasionally necessary, and at first glance, it seems simple. Example 9-17 shows one way to do it. However, as the remainder of this section shows, it's a bit more complex than it looks, and this is not actually the most efficient solution (which is another reason you might want to see if you can modify the design to avoid needing to do this in the first place).

Example 9-17. A delegate referring to another delegate

[2] Alternatively, you may just be one of nature's dynamic language enthusiasts, with an allergy to expressing semantics through static types. If that's the case, C# may not be the language for you, although check out C#'s dynamic features in Chapter 13 before deciding.

```
Predicate<string> pred = IsLongString;
var pred2 = new Func<string, bool>(pred);
```

The problem with Example 9-17 is that it adds an unnecessary level of indirection. The second delegate does not refer to the same method as the first one, it actually refers to the first delegate—so instead of a delegate that's a reference to `IsLongString`, the `pred2` variable ends up referring to delegate that is a reference to a delegate that is a reference to `IsLongString`. This is because the compiler treats Example 9-17 as though you had written the code in Example 9-18. (All delegate types have an `Invoke` method. It is implemented by the CLR, and it does the work necessary to invoke all of the methods to which the delegate refers.)

Example 9-18. A delegate explicitly referring to another delegate

```
Predicate<string> pred = IsLongString;
var pred2 = new Func<string, bool>(pred.Invoke);
```

In either Example 9-17 or Example 9-18, when you invoke the second delegate through the `pred2` variable, it will in turn invoke the delegate referred to by `pred`, which will end up invoking the `IsLongString` method. The right method gets called, just not as directly as we might like. If you know that the delegate refers to a single method (i.e., you're not using the multicast capability) Example 9-19 produces a more direct result.

Example 9-19. New delegate for the current target

```
Predicate<string> pred = IsLongString;
var pred2 = (Func<string, bool>) Delegate.CreateDelegate(
    typeof(Func<string, bool>), pred.Target, pred.Method);
```

This retrieves the target object and method from the `pred` delegate and uses it to create a new `Func<string, bool>` delegate. The result is a new delegate that refers directly to the same `IsLongString` method as `pred`. (The `Target` will be null because this is a static method, but I'm still passing it to `CreateDelegate` because I wanted to show code that works for both static and instance methods.) If you need to deal with multicast delegates, Example 9-19 won't work because it presumes that there's only one target method. You would need to call `CreateDelegate` in a similar way for each item in the invocation list. This isn't a scenario that comes up very often, but for completeness, Example 9-20 shows how it's done.

Example 9-20. Converting a multicast delegate

```
public static TResult DuplicateDelegateAs<TResult>(MulticastDelegate source)
{
    Delegate result = null;
    foreach (Delegate sourceItem in source.GetInvocationList())
    {
        var copy = Delegate.CreateDelegate(
            typeof(TResult), sourceItem.Target, sourceItem.Method);
        result = Delegate.Combine(result, copy);
    }

    return (TResult) (object) result;
}
```

In Example 9-20, the argument for the `TResult` type parameter has to be a delegate, so you may be wondering why I did not add a constraint for this type parameter. The obvious syntax to try would be `where`

> `TResult : delegate`. However, this doesn't work, and nor do the next two obvious choices: type constraints of `Delegate` or `MulticastDelegate`. Unfortunately, C# does not provide a way to write a constraint that requires a type argument to be a delegate.

These last few examples have depended upon various members of delegate types: `Invoke`, `Target` and `Method`. The last two of these come from the `Delegate` class, which is the base class of `MulticastDelegate`, from which all delegate types derive. The `Target` property's type is `object`. It will be null if the delegate refers to a static method, and otherwise, it will refer to the instance on which the method will be invoked. The `Method` property's type is `MethodInfo`. This is part of the reflection API, and it identifies a particular method. As Chapter 13 will show, you can use this to discover things about the method at runtime, but in the last two examples, we're just using it ensure that a new delegate refers to the same method as an existing one.

The third member, `Invoke`, is generated by the compiler. This is one of a few standard members that the C# compiler produces when you define a delegate type.

Behind the Syntax

Although as Example 9-3 showed, it takes just a single line of code to define a delegate type, the compiler turns this into a type that defines three methods and a constructor. Of course, the type also inherits members from its base classes. All delegates derive from `MulticastDelegate`, although all of the interesting instance members come from its base class, `Delegate`. (`Delegate` inherits from `object`, delegates all have the ubiquitous `object` methods too.) Even `GetInvocationList`, clearly a multicast-oriented feature, is defined by the `Delegate` base class.

> The split between `Delegate` and `MulticastDelegate` is the meaningless and arbitrary result of a historical accident. The original plan was to support both multicast and unicast delegates, but towards the end of the pre-release period for .NET 1.0 this distinction was dropped, and now all delegate types support multicast instances. This happened sufficiently late in the day that Microsoft felt it was too risky to merge the two base types into one, so the split remained even though it serves no purpose.

I've already shown all of the public instance members that `Delegate` defines. (`DynamicInvoke`, `GetInvocationList`, `Target` and `Method`.) Example 9-21 shows the signatures of the compiler-generated constructor and methods for a delegate type. The details vary from one type to the next; these are the generated members in the `Predicate<T>` type.

Example 9-21. The members of a delegate type

```
public Predicate(object target, IntPtr method)

public bool Invoke(T obj)

public IAsyncResult BeginInvoke(T obj, AsyncCallback callback, object state)
public bool EndInvoke(IAsyncResult result)
```

Any delegate type you define will have four similar members, and none of them will have bodies. The compiler generates the declarations, but the implementation is supplied automatically by the CLR.

The constructor takes the target object, which is null for static methods, and an `IntPtr` identifying the method. Notice that this is not the `MethodInfo` returned by the `Method` property. Instead, the constructor takes a *function token*, an opaque binary identifier for the target method. The CLR can provide binary metadata tokens for all members and types, but there's no C# syntax for working with them, so we don't normally see them. When you construct a new instance of a delegate type, the compiler automatically generates IL that fetches the function token. The reason delegates use tokens internally is that tokens can be more efficient than working with reflection API types such as `MethodInfo`.

The `Invoke` method is the one that calls the delegate's target method (or methods). You can use this explicitly from C#, as Example 9-22 shows. It is almost identical to Example 9-11, the only difference being that the delegate variable is followed by `.Invoke`. This generates exactly the same code as Example 9-11, so whether you write `Invoke`, or just use the syntax that treats delegate identifiers as though they were method names is a matter of style. As a former C++ developer, I've always felt at home with the Example 9-11 syntax, because it's similar to using function pointers in that language, but there's an argument that writing `Invoke` explicitly makes it easier to see that the code is using a delegate.

Example 9-22. Using Invoke explicitly

```
public static void CallMeRightBack(Predicate<int> userCallback)
{
    bool result = userCallback.Invoke(42);
    Console.WriteLine(result);
}
```

The `Invoke` method is the home for a delegate type's method signature. When you define a delegate type, this is where the return type and parameter list you specify end up. When the compiler needs to check whether a particular method is compatible with a delegate type (e.g., when you create a new delegate of that type) the compiler compares the `Invoke` method with the method you've supplied.

All delegate types have a pair of methods that offer asynchronous invocation. If you call `BeginInvoke`, the delegate will queue up a work item on the CLR's thread pool which will execute the target method. `BeginInvoke` returns without waiting for that invocation to complete (or even to begin). The `BeginInvoke` method's parameter list usually starts with all the same parameters as `Invoke`—just a single parameter of type `T` in the case of a `Predicate<T>`. If a delegate's signature has any `out` parameters, these will be omitted, because the method needs to run before it can return data through an `out` argument, and the whole point of `BeginInvoke` is that it doesn't wait for the method to complete. `BeginInvoke` adds two more parameters. The first is an `AsyncCallback`, which is a delegate type, and if you pass a non-null argument, the CLR will use this to call you back once the asynchronous execution has finished. The other argument is of type `object`, and whatever value you pass here will be handed back to you when the operation completes. The delegate doesn't do anything else with it—it's just for your benefit, and it can be a convenient way to keep track of which operation is which if multiple similar operations are in progress simultaneously.

The `EndInvoke` method provides a way to get the result of an operation launched with `BeginInvoke`. The delegate's return value becomes the return value of `EndInvoke`. We see `bool` here in Example 9-21 because that's the return type for `Predicate<T>`. If you define a delegate with any `out` or `ref` parameters, those will also show up on `EndInvoke`—anything that the method produces as a result goes here. If the operation throws an unhandled exception while running on the thread pool, the CLR catches and stores it, and rethrows it when you call `EndInvoke`. If you call `EndInvoke` before the operation completes, it will block, not returning until the operation finishes.

You can launch multiple simultaneous asynchronous operations against the same delegate, so `EndInvoke` needs some way of knowing which particular invocation you'd like to collect the results for. To enable this, `BeginInvoke` returns an `IAsyncResult`. This is an object that identifies a particular asynchronous operation in progress. If you ask to be notified when the operation is complete by supplying a non-null `AsyncCallback` argument to `BeginInvoke`, it passes this `IAsyncResult` to your completion callback. The `EndInvoke` takes an `IAsyncResult` as its argument, which is how it knows which invocation's results to return. `IAsyncResult` also defines an `AsyncState` property, which is where the final object argument you passed to `BeginInvoke` ends up.

> If you call `BeginInvoke`, it is mandatory that you make a corresponding call to `EndInvoke` at some point, even if there is no return value (or if there is but you don't care about it). Failure to call `EndInvoke` can cause the CLR to leak resources.

Using `BeginInvoke` and `EndInvoke` to run a delegate's target method on a thread pool thread is called *asynchronous delegate invocation*. (You'll also sometimes come across the inaccurate term "asynchronous delegates." That's a misnomer, because it implies that asynchronicity is a feature of the delegate. In fact, all delegates support both synchronous and asynchronous invocation, so this is a feature of how you use the delegate—it's the invocation that's asynchronous, not the delegate.) Although this was a popular way to perform asynchronous work with early versions of .NET, it's no longer so widely used, for three reasons. First, .NET 4.0 introduced the Task Parallel Library (TPL), which provides a more flexible and powerful abstraction for the services of the thread pool. Second, these methods implement an older pattern known as the Asynchronous Programming Model which does not fit directly with the new asynchronous language features of C#. Finally, the largest benefit of asynchronous delegate invocation was that it provided an easy way to pass a set of values from one thread to another—you could just pass whatever you needed as the arguments for the delegate—but this was obviated in C# 2.0 by the introduction of inline methods.

Inline Methods

C# lets you create delegates without needing to write a separate method explicitly, by defining an *inline method*, a method defined inside another method. (If the method returns a value, you'll also sometimes see it called an *anonymous function*.) For simple methods, this can remove a lot of clutter, but what makes this particularly useful is how it exploits the fact that delegates are more than just a reference to a method. Delegates can also include context, in the form of the target object for an instance method. The C#

compiler uses this to enable inline methods to get access to any variables that were in scope in the containing method at the point at which the inline method appears.

For historical reasons, C# provides two ways to define an inline method. The older way involves the `delegate` keyword, and is shown in Example 9-23. This form of inline method is known as an *anonymous method*.[3] I've put each argument for `FindIndex` on a separate line to make the inline method (the second argument) stand out, but C# does not require this.

Example 9-23. Anonymous method syntax

```
public static int GetIndexOfFirstNonEmptyBin(int[] bins)
{
    return Array.FindIndex(
        bins,
        delegate (int value) { return value > 0; }
    );
}
```

In some ways, this resembles the normal syntax for defining methods. The parameter list appears in parentheses, and is followed by a block containing the body of the method (which can contain as much code as you like by the way, and it is free to contain nested blocks, local variables, loops, and anything else you can put in a normal method). But instead of a method name, we just have the keyword `delegate`. The compiler infers the return type. In this case, it knows that `FindIndex` is expecting a `Predicate<T>` as the second argument, so it knows that the return type has to be `bool`.

In fact, the compiler knows more. I've passed `FindIndex` an `int[]` array, so the compiler knows that the type argument `T` is `int`, so we need a `Predicate<int>`. This means that in Example 9-23, I had to type in information—the type of the delegate's argument—that the compiler already knew. C# version 3.5, introduced a more compact inline method syntax that takes better advantage of what the compiler can deduce, shown in Example 9-24.

Example 9-24. Lambda syntax

```
public static int GetIndexOfFirstNonEmptyBin(int[] bins)
{
    return Array.FindIndex(
        bins,
        value => value > 0
    );
}
```

This form of inline method is called a *lambda expression*, and it is named after a branch of mathematics that is the basis of a function-based model for computation. There is no particular significance to the choice of the Greek letter lambda (λ). It was the accidental result of the limitations of 1930s printing technology. The inventor of lambda calculus, Alonzo Church, originally wanted a different notation, but when he came to published his first paper on the subject, the typesetting machine operator decided to print λ instead,

[3] Unhelpfully, there are two similar terms which somewhat arbitrarily mean almost but not quite the same thing. To clarify, the C# specification defines the term *anonymous function* as an alternative name for an *inline method* with a non-`void` return type, while an *anonymous method* is an inline method defined with the `delegate` keyboard.

because that was the closest available approximation to Church's notation that the machine could produce. Despite these inauspicious origins, this arbitrarily chosen term has become ubiquitous. LISP, an early and influential programming language, used the name lambda for expressions that are functions, and since then many languages have followed suit, including C#.

Example 9-24 is exactly equivalent to Example 9-23, I've just been able to leave various things out. The `=>` token unambiguously marks this out as being a lambda, so the compiler does not need that cumbersome and ugly `delegate` keyword just to recognize this as an inline method. The compiler knows that the method has to take an `int`, so there's no need to specify the parameter's type—I just provided the parameter's name: `value`. For simple methods that consist of just a single expression, the lambda syntax lets you omit the block and the `return` statement. This all makes for very compact lambdas, but in some cases, you might not want to omit quite so much, so as Example 9-25 shows, there are various optional features. Every lambda in that example is equivalent.

Example 9-25. Lambda variations

```
Predicate<int> p1 = value => value > 0;
Predicate<int> p2 = (value) => value > 0;
Predicate<int> p3 = (int value) => value > 0;
Predicate<int> p4 = value => { return value > 0; };
Predicate<int> p5 = (value) => { return value > 0; };
Predicate<int> p6 = (int value) => { return value > 0; };
```

The first variation is that you can put parentheses around the parameter. This is optional with a single parameter, but it is mandatory for multi-parameter lambdas. You can also be explicit about the parameters' types (in which case you will also need parentheses, even if there's only one parameter). And if you like, you can use a block instead of a single expression, at which point you also have to use the `return` keyword if the lambda returns a value. The normal reason for using a block would be if you wanted to write multiple statements inside the method.

You can also write a lambda that takes no arguments. As Example 9-26 shows, we just put an empty pair of parentheses in front of the `=>` token. (And as this example also shows, lambdas that use the greater than or equals operator, `>=`, can look a bit odd due the meaningless similarity between the `=>` and `>=` tokens.)

Example 9-26. A zero-argument lambda

```
Func<bool> isAfternoon = () => DateTime.Now.Hour >= 12;
```

The flexible and very compact syntax means that lambdas have all but displaced the older anonymous method syntax. However, the older syntax offers one advantage: it allows you to omit the argument list entirely. In some situations where you provide a callback, you only need to know that whatever you were waiting for has now happened. This is particularly common when using the standard event pattern described later in this chapter, because that requires event handlers to be passed arguments even in situations where they serve no purpose. For example, when a button is clicked, there's not much else to say beyond the fact that it was clicked, and yet all of the button types in .NET's various UI frameworks pass two arguments to the event handler. Example 9-27 successfully ignores this by using an anonymous method that omits the parameter list.

Example 9-27. Ignoring arguments in an anonymous method

```
EventHandler clickHandler = delegate { Debug.WriteLine("Clicked!"); };
```

`EventHandler` is a delegate type that requires its target methods to take two arguments, of type `object` and `EventArgs`. If our handler needed access to either, we could of course add a parameter list, but the anonymous method syntax lets us leave it out if we want. You cannot do this with a lambda.

Captured Variables

While inline methods often take up much less space than a full, normal method, they're not just about conciseness. The C# compiler uses a delegate's ability to refer not just to a method, but also to some additional context to provide an extremely useful feature: it can make variables from the containing method available to the inline method. Example 9-28 shows a method that returns a `Predicate<int>`. It creates this with a lambda that uses an argument from the containing method.

Example 9-28. Using a variable from the containing method

```
public Predicate<int> IsGreaterThan(int threshold)
{
    return value => value > threshold;
}
```

This provides the same functionality as the `ThresholdComparer` class from Example 9-7, but it now achieves it in a single, simple method, rather than needing to write an entire class. In fact the code is almost deceptively simple, so it's worth looking closely at what it does. The `IsGreaterThan` method returns a delegate instance. That delegate's target method performs a simple comparison—it evaluates the `value > threshold` expression and returns the result. The `value` variable in that expression is just the delegate's argument—the `int` passed by whichever code invokes the `Predicate<int>` that `IsGreaterThan` returns. The second line of Example 9-29, invokes that code passing in 200 as the argument for `value`.

Example 9-29. Where value comes from

```
Predicate<int> greaterThanTen = IsGreaterThan(10);
bool result = greaterThanTen(200);
```

The `threshold` variable in the expression is trickier. This is not an argument to the inline method. It's the argument of `IsGreaterThan`, and Example 9-29 passes a value of 10 as the `threshold` argument. However, `IsGreaterThan` has to return before we can invoke the delegate it returns. If the method for which that `threshold` variable was an argument has already returned, you might think that the variable would no longer be available by the time we invoke the delegate. In fact, it's fine because the compiler does some work on our behalf. If an inline method uses any arguments, or any local variables that were declared by the containing method, the compiler generates a class to hold those variables so that they can outlive the method that created them. This is one of the reasons that the popular myth that says local variables of value type live on the stack is not always true—in this case, the compiler copies the incoming `threshold` argument's value to a field of an object on the heap, and any code that uses the `threshold` variable ends up using that field instead. Example 9-30 shows the generated code that the compiler produces for the inline method in Example 9-28.

Example 9-30. Code generated for an inline method

```
[CompilerGenerated]
private sealed class <>c__DisplayClass1
```

```
{
    public int threshold;

    public bool <IsGreaterThan>b__0(int value)
    {
        return (value > this.threshold);
    }
}
```

The class and method names all begin with characters that are illegal in C# identifiers, to ensure that this compiler-generated code cannot clash with anything we write. (The exact names are not fixed by the way—you may find they are slightly different if you try this.) This generated code bears a striking resemblance to the `ThresholdComparer` class from Example 9-7, which is unsurprising, because the goal is the same: the delegate needs some method that it can refer to, and that method's behavior depends on a value that is not fixed. Inline methods are not a feature of the runtime's type system, so the compiler has to generate a class to provide this kind of behavior on top of the CLR's basic delegate functionality.

Once you know that this is what's really happening when you write an inline method, it follows naturally that the inner method is able not just to read the variable, but also to modify it. This variable is just a field in an object that two methods—the inline method and the containing method—have access to. Example 9-31 uses this to maintain a count that is updated from an inline method.

Example 9-31. Modifying a captured variable

```
static void Calculate(int[] nums)
{
    int zeroCount = 0;
    int[] nonZeroNums = Array.FindAll(
        nums,
        v =>
        {
            if (v == 0)
            {
                zeroCount += 1;
                return false;
            }
            else
            {
                return true;
            }
        });
    Console.WriteLine(
        "Number of zero entries: {0}, first non-zero entry: {1}",
        zeroCount,
        nonZeroNums[0]);
}
```

Everything in scope for the containing method is also in scope for inline methods. If the containing method is an instance method, this includes any instance members of the type, so your inline method could access fields, properties and methods. (The compiler supports this by adding a field to the generated class to hold a copy of the `this` reference.) The compiler only puts what it needs to in generated classes of the kind shown in Example 9-30, and if you don't use any variables or instance members from the

containing scope, it might not even have to generate a class at all, and may be able to generate just a method.

The `FindAll` method in the preceding examples does not hold onto the delegate after it returns—any callbacks will happen while `FindAll` runs. Not everything works that way though. Some APIs perform asynchronous work, and will call you back at some point in the future, by which time the containing method may have returned. This means that any variables captured by the inline method will live longer than the containing method. In general, this is fine because all of the captured variables live in an object on the heap, so it's not as though the inline method is relying on a stack frame that is no longer present. The one thing you need to be careful of though is explicitly releasing resources before callbacks have finished. Example 9-32 shows an easy mistake to make. This uses an asynchronous, callback-based API to discover the HTTP content type of the resource at a particular URL. (The `BeginGetResponse` and `EndGetResponse` methods in this example use a very similar pattern to the `BeginInvoke` and `EndInvoke` delegate methods I described earlier, incidentally.)

Example 9-32. Premature disposal

```
using (var file = new StreamWriter(@"c:\temp\log.txt"))
{
    var req = WebRequest.Create("http://www.interact-sw.co.uk/");
    req.BeginGetResponse(iar =>
    {
        var resp = req.EndGetResponse(iar);

        // BAD! This StreamWriter will probably have been disposed
        file.WriteLine(resp.ContentType);
    }, null);

} // Will probably dispose StreamWriter before callback runs
```

The `using` statement in this example will dispose the `StreamWriter` as soon as execution reaches the point at which the `file` variable goes out of scope in the outer method. The problem is that this `file` variable is also used in an inner method which will in all likelihood run after the thread executing that outer method has left that using statement's block. The problem is that the compiler has no understanding of when the inner block will run—it doesn't know whether that's a synchronous callback like `Array.FindAll` uses, or an asynchronous one. So it cannot do anything special here—it just calls `Dispose` at the end of the block because that's what our code told it to do. In practice, a `using` statement is not a good choice here—I would need to write code to dispose the stream writer explicitly at a point where I could be certain that I have finished with it.

> The new asynchronous language features, discussed in Chapter 18, can help avoid this sort of problem. You use that in conjunction with APIs that present a particular pattern that makes it possible for the compiler to know exactly how long things remain in scope. The constraints imposed by that pattern make it possible for a `using` statement to call `Dispose` at the correct moment.

In performance-critical code you may need to bear the costs of inline methods in mind. If the inline method uses any variables from the outer scope, then each time you create a delegate to refer to the inline method, you may be creating two objects instead of one: a

delegate instance and an instance of the generated class to hold shared local variables. The compiler will reuse these variable holders when it can—if one method contains two inline methods, they may be able to share an object, for example. Even with this sort of optimization, you're still creating additional objects, increasing the pressure on the garbage collector. It's not particularly expensive—these are typically small objects—but if you're up against a particularly oppressive performance problem, you might be able to eke out some small improvements by writing things in a more long-winded fashion to be able to reduce the number of object allocations.

Variable capture can also occasionally lead to bugs, particularly due to a subtle scope-related issue with `for` and `foreach` loops. In fact, this was sufficiently easy to run into that Microsoft has changed how `foreach` behaves in the most recent version of C#. The issue still exists with `for`, and Example 9-33 runs into it.

Example 9-33. Problematic variable capture in a for loop

```
static void Main(string[] args)
{
    var greaterThanN = new Predicate<int>[10];
    for (int i = 0; i < greaterThanN.Length; ++i)
    {
        greaterThanN[i] = value => value > i; // Bad use of i
    }

    Console.WriteLine(greaterThanN[5](20));
    Console.WriteLine(greaterThanN[5](6));
}
```

This example initializes an array of `Predicate<int>` delegates, where each delegate tests whether the value is greater than some number. Specifically, it compares the value with `i`, the loop counter that decides where in the array each delegate goes, so you might expect the element at index 5 to refer to a method that compares its argument with 5. If that were so, this program would print out True twice. In fact it prints out True and then False. It turns out that Example 9-33 produces an array of delegates where every single element compares its argument with 10.

This usually surprises people when they encounter it. With hindsight, it's easy enough to see why this happens when you know how the C# compiler enables a lambda to use variables from its containing scope. The `for` loop declares the `i` variable, and because it is used both by the containing `Main` method and each delegate the loop creates, the compiler will generate class similar to the one in Example 9-30, and the variable will live in a field of that class. Since the variable comes into scope when the loop starts, and remains in scope for the duration of the loop, the compiler will create one instance of that generated class, and it will be shared by all of the delegates. So as the loop increments `i`, this modifies the behavior of all of the delegates, because they all use that same `i` variable.

Fundamentally, the problem is that there's only one `i` variable here. You can fix the code by introducing a new variable inside the loop. Example 9-34 copies the value of `i` into another local variable, `current`, which does not come into scope until an iteration is underway, and which goes out of scope at the end of each iteration. So although there is only one `i` variable, which lasts for as long as the loop runs, we get what is effectively a new `current` variable each time round the loop. Because each delegate gets its own distinct `current` variable, this modification means that each delegate in the array

compares with a different value, the value that the loop counter had for that particular iteration.

Example 9-34. Modifying a loop to capture the current value

```
for (int i = 0; i < greaterThanN.Length; ++i)
{
    int current = i;
    greaterThanN[i] = value => value > current;
}
```

The compiler still generates a class similar to the one in Example 9-30 to hold the `current` variable that's shared by the inline and containing methods, but this time it will create a new instance of that class each time round the loop, to be able give each inline method a different instance of that variable.

You may be wondering what would happen if you wrote an inline method that used variables at multiple scopes. Example 9-35 declares a variable called `offset` before the loop, and the lambda uses both that and a variable whose scope lasts for only one iteration.

Example 9-35. Capturing variables at different scopes

```
int offset = 0;
for (int i = 0; i < greaterThanN.Length; ++i)
{
    int current = i;
    greaterThanN[i] = value => value > (current + offset);
}
```

In that case, the compiler would generate two classes, one to hold any per-iteration shared variables (`current`, in this example) and one to hold those whose scope spans the whole loop (`offset`, in this case). Each delegate's target object would be the object containing inner scope variables, and that would contain a reference to the outer scope.

Figure 9-1 shows roughly how this would work, although it has been simplified to show just the first five items. The `greaterThanN` variable contains a reference to an array. Each array element contains a reference to a delegate. Each delegate refers to the same method, but each one has a different target object, which is how each delegate can capture a different instance of the `current` variable. Each of these target objects refers to a single object containing the `offset` variable captured from the scope outside of the loop.

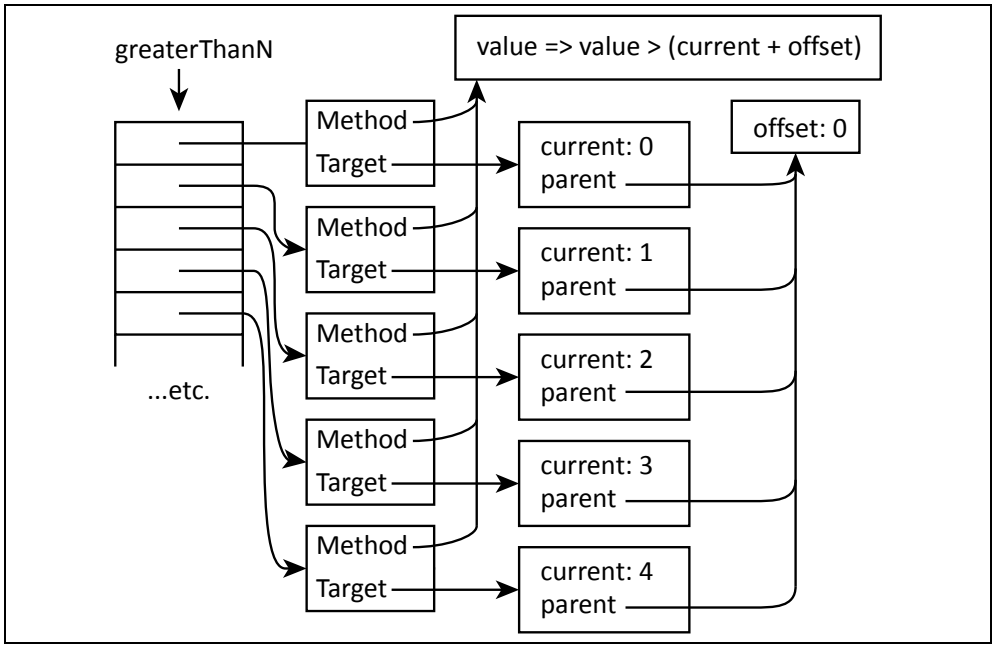

Figure 9-1. Figure Caption Text Goes Here

In versions of C# up to and including v4.0, `foreach` loops worked in way that would cause the same problem, and which needed a similar extra local variable to fix. The iteration variable came into scope before the first iteration and remained in scope for the whole loop, changing its value at each iteration, leading to the same potential problem as a `for` loop. But this has changed: it is now as though a new iteration variable comes into scope each time around the loop, so if you capture that variable in an inline method, you get the value for that iteration, not the value of the most recent iteration to have started.

This change will break any code that relied on the original behavior. However, the original behavior was not very useful, and was a frequent cause of bugs, so Microsoft felt it was worth making the change. They left the `for` loop's behavior unchanged because that construct leaves more of the work of iteration to the developer—it just provides placeholders for initialization, loop termination testing, and iteration, so unlike with a `foreach` loop, it's not always clear what would count as the iteration variable. An example such as `for (var x = new Item(); !file.EndOfStream; source.Next())` is legal, and it's not clear which identifier, if any, should get special treatment. So `for` loops continue to work as they always have.

Lambdas and Expression Trees

Lambdas have an additional trick up their sleeves beyond providing delegates. Some lambdas can produce a data structure that represents code. This occurs when you use the lambda syntax in a context that requires an `Expression<T>` where `T` is a delegate type. `Expression<T>` itself is not a delegate type; it is a special type in the .NET Framework class library (in the `System.Linq.Expressions` namespace) that triggers this alternative handling of lambdas in the compiler. Example 9-36 uses this type.

Example 9-36. A lambda expression

```
Expression<Func<int, bool>> greaterThanZero = value => value > 0;
```

This example looks very similar to some of the lambdas and delegates I've shown already in this chapter, but the compiler handles this very differently. It will not generate a method—there will be no compiled IL representing the lambda's body. Instead, the compiler will produce code similar to that in Example 9-37.

Example 9-37. What the compiler does with a lambda expression

```
ParameterExpression valueParam = Expression.Parameter(typeof(int), "value");
ConstantExpression constantZero = Expression.Constant(0);
BinaryExpression comparison = Expression.GreaterThan(valueParam, constantZero);
Expression<Func<int, bool>> greaterThanZero =
    Expression.Lambda<Func<int, bool>>(comparison, valueParam);
```

This code calls various factory functions provided by the `Expression` class to produce an object for each sub-expression in the lambda. This starts with the simple operands—the `value` parameter and the constant value 0. These are fed into an object representing the 'greater than' comparison expression, which in turn becomes the body of an object representing the whole lambda expression.

The ability to produce an object model for an expression makes it possible to write an API where the behavior is controlled by the structure and content of an expression. For example, some data access APIs can take an expression similar to the ones produced by either Example 9-36 or Example 9-37 and use it to generate part of a database query. I'll be talking about C#'s integrated query features in Chapter 10, and data access in Chapter 19 but Example 9-38 gives a flavor of how a lambda expression can be used as the basis of a query.

Example 9-38. Expressions and database queries

```
var expensiveProducts = dbContext.Products.Where(p => p.ListPrice > 3000);
```

This example happens to use a .NET feature called the Entity Framework, but other data access technologies support the same approach. The `Where` method in this example takes an argument of type `Expression<Func<Product,bool>>`.[4] `Product` is a class that corresponds to an entity in the database, but the important part here is the use of `Expression<T>`. That means that the compiler will generate code that creates a tree of objects whose structure corresponds to that lambda expression. The `Where` method processes that expression tree, generating a SQL query which includes this clause: `WHERE [Extent1].[ListPrice] > cast(3000 as decimal(18))`. So although I wrote my query as a C# expression, the work required to find matching objects will all happen on my database server.

Lambda expressions were added to C# to enable this sort of query handling as part of the set of features known collectively as LINQ (which is the subject of Chapter 10). However, as with most LINQ-related features it's possible to use them for other things. For example, at http://www.interact-sw.co.uk/iangblog/2008/04/13/member-lifting you'll find code that takes expressions that retrieve properties, e.g. `obj.Prop1.Prop2`, and

[4] You may be surprised to see `Func<Product,bool>` here and not `Predicate<Product>`. The `Where` method is part of a .NET feature called LINQ that makes extensive use of delegates. To avoid defining huge numbers of new delegate types, LINQ uses `Func` types, and for consistency across the API, it prefers `Func` even when other standard types are available.

modifies them to tolerate nulls. If either `obj` or `obj.Prop1` were null, evaluating that expression would normally produce a `NullReferenceException`, but it's possible to transform this into an expression that evaluates to null if a null is encountered at any stage. However, I'm not convinced the benefits of this sort of expression tinkering necessarily outweigh the problems it causes—I wrote that null tolerance example as a learning exercise, and what it taught me was that this particular kind of 'clever' code is more trouble than it's worth. (That's why I've not shown an equivalent example in this book—it's a lot of code, and it offers rather dubious benefits.) When it comes to production code, I've only ever used expression trees in conjunction with LINQ, the scenario for which they were designed. My experience with them in other areas is that the complexity tends to end up with code that's painful to maintain. That's not to say that you should absolutely avoid it, just that you should be wary. The expense might be worthwhile for companies like Microsoft, who are producing frameworks used by millions of developers, with a budget to match, but if that doesn't describe your project, you might want to think twice before inflicting the awesome coolness of your expression tree wrangling on your customers.

Events

Delegates provide the basic callback mechanism required for notifications, but there are many mays you could go about using them. Should the delegate be passed as a method argument, a constructor argument, or perhaps as a property? How should you support unsubscribing from notifications? The CTS formalizes the answers to these questions through a special kind of class member called an *event*, and C# has syntax for working with events. Example 9-39 shows a class with one event member.

Example 9-39. A class with an event

```
public class Eventful
{
    public event Action<string> Announcement;

    public void Announce(string message)
    {
        if (Announcement != null)
        {
            Announcement(message);
        }
    }
}
```

As with all members, you can start with an accessibility specifier, and it will default to private if you leave that off. Next, the `event` keyword singles this out as an event. Then there's the event's type, which can be any delegate type. I've used `Action<string>`, although as you'll soon see, this is an unorthodox choice. Finally, we put the member name, so this example defines an event called `Announcement`.

To handle an event, you must provide a delegate of the right type, and you must use the `+=` syntax to attach that delegate as the handler. Example 9-40 uses a lambda, but you can use any expression that produces, or is implicitly convertible to a delegate of the type the event requires.

Example 9-40. Handling events

```
var source = new Eventful();
source.Announcement += m => Console.WriteLine("Announcement: " + m);
```

Example 9-39 also shows how to *raise* an event, i.e., how to invoke all the handlers that have been attached to the event. Its `Announce` method checks the event member to see if it is null, and if not, uses the same syntax we would use if `Announcement` were a field containing a delegate that we wanted to invoke. In fact, as far as code inside the class is concerned, that's exactly what an event looks like—it appears to be a field containing a delegate.

So why do we need a special member type if this looks just like a field? Well it only looks like a field from inside the defining class. Code outside of the class cannot raise the event, so the code shown in Example 9-41 will not compile.

Example 9-41. How not to raise an event

```
var source = new Eventful();
source.Announcement("Will this work?"); // No, this will not even compile
```

From the outside, the only things you can do to an event are to attach a handler using `+=` and to remove one using `-=`. The syntax for adding and removing event handlers is unusual, in that it's the only case in C# in which you get to use `+=` and `-=` without the corresponding standalone `+` or `-` operators being available. The actions performed by `+=` and `-=` on events both turn out to be method calls in disguise. Just as properties are really pairs of methods with a special syntax, so are events. They are similar in concept to the code shown in Example 9-42. (In fact, the real code includes some moderately complex lock-free thread-safe code. I've not shown this because the multithreading obscures the basic intent.) This won't have quite the same effect, because the `event` keyword adds metadata to the type identifying the methods as being an event, so this is just for illustration.

Example 9-42. The approximate effect of declaring an event

```
private Action<string> Announcement;

// Not the actual code.
// The real code is more complex, to tolerate concurrent calls
public void add_Announcement(Action<string> handler)
{
    Announcement += handler;
}
public void remove_Announcement(Action<string> handler)
{
    Announcement -= handler;
}
```

Just as with properties, events exist mainly to offer a convenient, distinctive syntax, and to make it easier for tools to know how to present the features that classes offer. Events are particularly important for user interface elements. In most UI frameworks, the objects representing interactive elements can often raise a wide range of events, corresponding to various forms of input such as keyboard, mouse, or touch. There are also often events relating to behavior specific to a particular control, such as selecting a new item in a list. Because the CTS defines a standard idiom by which elements can expose events, visual UI designers such as the ones built into Visual Studio can display the available events, and offer to generate handlers for you.

Standard Event Delegate Pattern

The event in Example 9-39 is unusual, in that it uses the `Action<T>` delegate type. This is perfectly legal, but in practice, you will rarely see that, because almost all events use delegate types that conform to a particular pattern. This pattern requires the delegate's method signature to have two arguments. The first argument's type is `object`, and the second's type is either `EventArgs`, or some type derived from `EventArgs`. Example 9-43 shows the `EventHandler` delegate type in the `System` namespace, which is the simplest and most widely used example of this pattern.

Example 9-43. The EventHandler delegate type

```
public delegate void EventHandler(object sender, EventArgs e);
```

The first argument is usually called `sender`, because the event source passes a reference to itself for this argument. This means that if you attach a single handler method to multiple event sources, that handler can always know which source raised any particular notification.

The second argument provides a place to put information specific to the event. For example, WPF user interface elements define various events for handling mouse input that use more specialized delegate types, such as `MouseButtonEventHandler`, with signatures that specify a corresponding specialized event argument offering details about the event. For example, `MouseButtonEventArgs` defines a `GetPosition` method that tells you where the mouse was when the button was clicked, and it defines various other properties offering further detail, including `ClickCount`, and `Timestamp`.

Whatever the specialized type of the second argument may be, it will always derive from the base `EventArgs` type. That base type is not very interesting—it does not add any members beyond the standard ones provided by `object`. However, it does make it possible to write a general purpose method that can be attached to any event that uses this pattern. The rules for delegate compatibility mean that even if the delegate type specifies a second argument of type `MouseButtonEventArgs`, a method whose second argument is of type `EventArgs` is an acceptable target. This can occasionally be useful for code generation or other infrastructure scenarios. However the main benefit of the standard event pattern is simply one of familiarity—experienced C# developers generally expect events to work this way.

Custom Add and Remove Methods

Sometimes, you might not want to use the default event implementation generated by the C# compiler. For example, a class may define a large number of events, most of which will not be used on the majority of instances. User interface frameworks often have this characteristic. A WPF user interface can have thousands of elements, every one of which offers over a hundred events, but you normally only attach handlers to a few of these elements, and even with these, you only handle a fraction of the events on offer. It is inefficient for every element to dedicate a field to every available event in this case.

Using the default field-based implementation for large numbers of rarely-used events could add hundreds of bytes to the footprint of each element in a user interface, which can have a discernible effect on performance. (In WPF, this could add up to a few hundred thousand bytes. That might not sound like much given modern computers' memory capacities, but it can put your code in a place where it is no longer able to make

efficient use of the CPU's cache, causing a nosedive in application responsiveness. Even if the cache is several megabytes in size, the fastest parts of the cache are usually much smaller, and wasting a few hundred kilobytes in a critical data structure can make a world of different to performance.)

Another reason you might want to eschew the default compiler-generated event implementation is that you may want more sophisticated semantics when raising events. For example, WPF supports event bubbling: if a UI element does not handle certain events, they will be offered to the parent element, then the parent's parent, and so on up the tree until a handler is found or it reaches the top. Although it would be possible to implement this sort of scheme with the standard event implementation C# supplies, much more efficient strategies are possible when event handlers are relatively sparse.

To support these scenarios, C# lets you provide your own add and remove methods for an event. It will look just like a normal event from the outside—anyone using your class will use the same `+=` and `-=` syntax to add and remove handlers, and it will not be possible to tell that it provides a custom implementation. Example 9-44 shows a class with two events, and it uses a single dictionary, shared across all instances of the class, to keep track of which events have been handled on which objects. The approach is extensible to larger numbers of events—the dictionary uses pairs of objects as the key, so each entry represents a particular (source, event) pair. (It's not safe for multi-threaded use by the way. This example just illustrates how custom event handlers look—it's not a fully-engineered solution.)

Example 9-44. Custom add and remove for sparse events

```
public class ScarceEventSource
{
    // One dictionary shared by all instances of this class,
    // tracking all handlers for all events.
    private static readonly
     Dictionary<Tuple<ScarceEventSource, object>, EventHandler> _eventHandlers
      = new Dictionary<Tuple<ScarceEventSource, object>, EventHandler>();

    // Objects used as keys to identify particular events in the dictionary.
    private static readonly object EventOneId = new object();
    private static readonly object EventTwoId = new object();


    public event EventHandler EventOne
    {
        add
        {
            AddEvent(EventOneId, value);
        }
        remove
        {
            RemoveEvent(EventOneId, value);
        }
    }

    public event EventHandler EventTwo
    {
        add
        {
            AddEvent(EventTwoId, value);
```

```
        }
        remove
        {
            RemoveEvent(EventTwoId, value);
        }
    }

    public void RaiseBoth()
    {
        RaiseEvent(EventOneId, EventArgs.Empty);
        RaiseEvent(EventTwoId, EventArgs.Empty);
    }

    private Tuple<ScarceEventSource, object> MakeKey(object eventId)
    {
        return Tuple.Create(this, eventId);
    }

    private void AddEvent(object eventId, EventHandler handler)
    {
        var key = MakeKey(eventId);
        EventHandler entry;
        _eventHandlers.TryGetValue(key, out entry);
        entry += handler;
        _eventHandlers[key] = entry;
    }

    private void RemoveEvent(object eventId, EventHandler handler)
    {
        var key = MakeKey(eventId);
        EventHandler entry = _eventHandlers[key];
        entry -= handler;
        if (entry == null)
        {
            _eventHandlers.Remove(key);
        }
        else
        {
            _eventHandlers[key] = entry;
        }
    }

    private void RaiseEvent(object eventId, EventArgs e)
    {
        var key = MakeKey(eventId);
        EventHandler handler;
        if (_eventHandlers.TryGetValue(key, out handler))
        {
            handler(this, e);
        }
    }
}
```

The syntax for custom events is reminiscent of the full property syntax: we add a block after the member declaration that contains the two members, which are called `add` and `remove` instead of `get` and `set`. (Unlike with properties, you are required to supply both methods.) This disables the generation of the field that would normally hold the

event, meaning that the `ScarceEventSource` class has no instance fields at all—instances of this type are as small as it's possible for an object to be.

The price for this small memory footprint is a considerable increase in complexity—I've written about 16 times as many lines of code as I would have needed with compiler-generated events. Moreover, this technique only provides an improvement if the events really are not handled most of the time—if I attached handlers to both events for every instance of this class, the dictionary-based storage would consume more memory than simply having a field for each event in each instance of the class. So you should only consider this sort of custom event handling if you either need non-standard event raising behavior, or if you are very sure that you really will be saving memory, and that the savings are worthwhile.

Events and the Garbage Collector

As far as the garbage collector is concerned, delegates are normal objects like any other. If the GC discovers that a delegate instance is reachable, then it will inspect the `Target` property, and whichever object that refers to will also be considered reachable, along with whatever objects that object in turn refers to. Although there is nothing remarkable about this, there are situations in which leaving event handlers attached can cause objects to hang around in memory when you might have expected them to be collected by the GC.

There's nothing intrinsic to delegates and events that makes them unusually likely to defeat the garbage collector. If you do get an event-related memory leak, it will have the same structure as any other .NET memory leak: starting from a root reference, there will be some chain of references that keeps an object reachable even after you've finished using it. The only reason events get special blame for memory leaks is that they are often used in ways that can cause problems.

For example, suppose your application maintains some object model representing its state, and that your user interface code is in a separate layer that makes use of that underlying model, adapting the information it contains for presentation on screen. This sort of layering is usually advisable—it's a bad idea to intermingle code that deals with user interactions and code that implements the application's logic. But a problem can arise if the underlying model advertises changes in state that the UI needs to reflect. If these changes are advertised through events, your UI code will typically attach handlers to those events.

Now imagine that someone closes one of your application's windows. You would hope that the objects representing that window's user interface will all be detected as unreachable the next time the GC runs. The UI framework is likely to have attempted to make that possible. For example, WPF ensures that each instance of its `Window` class is reachable for as long as the corresponding window is open, but once the window has been closed, it stops holding any references to the window, to enable all of the UI objects for that window to be collected.

However, if you handle an event from your main application's model with a method in a `Window`-derived class, and if you do not explicitly remove that handler when the window is closed, you will have a problem. As long as your application is still running, something somewhere will presumably be keeping your application's underlying model reachable. This means that the target objects of any delegates held by your application model (e.g., delegates that were added as event handlers) will continue to be reachable,

preventing the garbage collector from freeing them. So if a `Window`-derived object for the now-closed window is still handling events from your application model, that window, and all of the UI elements it contains will still be reachable, and will not be garbage collected.

> There's a persistent myth that this sort of event-based memory leak has something to do with circular references. The garbage collector copes perfectly well with circular references. It's true that there are circular references in these scenarios, but they're not the issue. The problem is caused by accidentally keeping objects reachable after you no longer need them. Doing that will cause problems regardless of whether circular references are present.

There are a couple of ways you can deal with this. One is simply to ensure that if your UI layer ever attaches handlers to objects that will stay alive for a long time, you should make sure that you remove those handlers when the relevant UI element is no longer in use. Alternatively, you could use weak references to ensure that if your event source is the only thing holding a reference to the target, it doesn't keep it alive. WPF can help you with this—it provides a `WeakEventManager` class that provides a way to handle an event in such a way that the handling object is able to be garbage collected without needing to unsubscribe from the event. WPF uses this technique itself when databinding the UI to a data source that provides property change notification events.

> Although event-related leaks often arise in user interfaces, they can occur anywhere. As long as an event source remains reachable, all of its attached handlers will also remain reachable.

Events vs. Delegates

Some APIs provide notifications through events, while others just use delegates directly. How should you decide which approach to use? In some cases, the decision may be made for you because you want to support some particular idiom. For example, if you want your API to support the new asynchronous features in C#, you will need to implement the pattern described in Chapter 18, which involves taking a delegate as a method argument. If you are writing a user interface element, events will most likely be appropriate because that's the predominant idiom.

In cases where constraints or conventions do not provide an answer, you need to think about how the callback will be used. If there will be multiple subscribers for a notification, an event could be the best choice. This is not absolutely necessary because any delegate is capable of multicast behavior, but by convention, this behavior is usually offered through events. If users of your class will need to remove the handler at some point, events are also likely to be a good choice. Having said that, the `IObservable<T>` interface might be a better choice if you need more advanced functionality. This interface is part of the Reactive Extensions for .NET, and is described in Chapter 11.

You would typically pass a delegate as a method or constructor argument if it only makes sense to have a single target method. For example, if the delegate type has a non-`void` return value which the API depends on (such as the `bool` returned by the predicate passed to `Array.FindAll`) it makes no sense to have multiple targets, or zero targets.

An event is the wrong idiom here, because it's perfectly normal to attach either no handlers or multiple handlers to an event.

Occasionally it might make sense to have either zero or one handler, but never more than one. For example, WPF's `CollectionView` class can sort, group, and filter data from a collection. You configure filtering by providing a `Predicate<object>`. This is not passed as a constructor argument because filtering is optional, so instead, the class defines a `Filter` property. An event would be inappropriate here, partly because `Predicate<object>` does not fit the usual event delegate pattern, but mainly because the class needs an unambiguous answer of yes or no, so it does not want to support multiple targets. (The fact that all delegate types support multicast means that it's still possible to supply multiple targets of course. But the decision to use a property rather than event signals the fact that it's not useful to attempt to provide multiple callbacks here.)

Delegates vs Interfaces

Back at the start of this chapter, I argued that delegates offer a less cumbersome mechanism for callbacks and notifications than interfaces. So why do some APIs require callers to implement an interface to enable callbacks? Why do we have `IComparer<T>` and not a delegate? Actually we have both—there's a delegate type called `Comparison<T>`, which is supported as an alternative by many of the APIs that accept an `IComparer<T>`. Arrays and `List<T>` have overloads of their `Sort` methods taking either.

There are some situations in which the object-oriented approach may be preferable to using delegates. An object that implements `IComparer<T>` could provide properties to adjust the way the comparison works. Some callbacks may want to collect and summarize information across multiple callbacks, and although you can do that through captured variables, it may be easier to get the information back out again at the end if it's available through properties of an object.

This is really a decision for whoever is writing the code being called back, and not for the developer writing the code that makes the call. Delegates ultimately are more flexible because they allow the consumer of the API to decide how to structure their code, whereas an interface imposes constraints. However, if an interface happens to align with the abstractions you want, delegates can seem like an irritating extra detail. This is why some APIs present both options, such as the sorting APIs that accept either an `IComparer<T>` or a `Comparison<T>`.

One situation in which interfaces might be preferable to delegates is if you need to provide multiple related callbacks. The Reactive Extensions for .NET (see Chapter 11) define an abstraction for notifications that include the ability to know when you've reached the end of a sequence of events, or when there has been an error, so in that model, subscribers implement an interface with three methods—`OnNext`, `OnCompleted` and `OnError`. It makes sense to use an interface because all three methods are typically required for a complete subscription.

Summary

Delegates are objects that provide a reference to a method, which can be either static or an instance method. With instance methods, the delegate also holds a reference to the target object, so the code that invokes the delegate does not need to supply a target. Delegates can also refer to multiple methods, although that complicates matters if the delegate's return type is not `void`. Although delegate types get special handling from the CLR, they are still just a reference type, meaning that a reference to a delegate can be passed as an argument, returned from a method, and stored in a field, variable or property. A delegate type defines a signature for the target method. This is actually represented through the type's `Invoke` method, but C# can hide this, offering a syntax in which you can invoke a delegate expression directly without explicitly referring to `Invoke`. You can construct a delegate that refers to any method with a compatible signature. You can also get C# to do more of the work for you—if you write an inline method, C# will supply a suitable declaration for you, and can also do work behind the scenes to make variables in the containing method available to the inner one. Delegates are the basis of events, which provide a formalized publish/subscribe model for notifications.

One C# feature that makes particularly extensive use of delegates is LINQ, which is the subject of the next chapter.

10

LINQ

Language Integrated Query (LINQ) is a powerful set of tools for working with sets of information in C#. It is useful in any application that needs to work with multiple pieces of data (i.e., almost any application). Although one of its primary goals was to provide straightforward access to relational databases, LINQ is applicable to many kinds of information. For example, it can also be applied to in-memory object models, HTTP-based information services, and XML documents.

LINQ is not a single feature. It relies on several language elements that work together. The most conspicuous LINQ-related language feature is the *query expression*, a form of expression that loosely resembles a database query, but which can be used to perform queries against any supported source, including plain old objects. As you'll see, query expressions rely heavily on some other language features such as lambdas, extension methods, and expression object models.

Language support is only half the story. LINQ needs class libraries to implement a standard set of querying primitives called *LINQ operators*. Each different kind of data requires its own implementation, and a set of operators for any particular type of information is referred to as a *LINQ provider*. The .NET Framework class library has several built-in providers, including one for working directly with objects (called LINQ to Objects), and a couple for working with databases (LINQ to SQL, which is specific to SQL Server, and the more complex but more general-purpose LINQ to Entities). The WCF Data Services client library for consuming OData-based web services also has a LINQ provider. In short, LINQ is a widely supported idiom in the .NET Framework, and it's extensible, so you will also find open source and other third party providers.

Most of the examples in this chapter use LINQ to Objects. This is partly because it avoids cluttering the examples with extraneous details such as database or service connections, but there's a more important reason. LINQ's introduction in 2007 significantly changed the way I write C#, and that's entirely because of LINQ to Objects. Although LINQ's syntax makes it look like it's primarily a data access technology, I have found it to be far more useful than that. Having LINQ's services available on any collection of objects makes it useful in every part of your code.

Query Expressions

The most visible feature of LINQ is the query expression syntax. It's not the most important—as we'll see later, it's entirely possible to use LINQ productively without ever writing a query expression. However, it's a very natural syntax for many kinds of queries, so it takes center stage despite being technically optional.

At first glance, a query expression loosely resembles a database query, but the syntax works with any LINQ provider. Example 10-1 shows a query expression that uses LINQ to Objects to search for certain `CultureInfo` objects. (A `CultureInfo` provides a set of culture-specific information, such as the symbol used for the local currency, what language is spoken, and so on. Some systems call this a locale.) This particular query looks at the character that denotes what would, in English, be called the decimal point. Many countries actually use a comma instead of a period, and in those countries 100,000 would mean the number one hundred written out to three decimal places; in English-speaking cultures we would normally write this as 100.000. The query expression searches all the cultures known to the system, and returns the ones that use a comma as the decimal separator.

Example 10-1. A LINQ query expression

```
IEnumerable<CultureInfo> commaCultures =
    from culture in CultureInfo.GetCultures(CultureTypes.AllCultures)
    where culture.NumberFormat.NumberDecimalSeparator == ","
    select culture;

foreach (CultureInfo culture in commaCultures)
{
    Console.WriteLine(culture.Name);
}
```

The `foreach` loop in this example shows the results of the query. On my system, this prints out the name of 187 cultures, indicating that slightly over half of the 354 available cultures use a comma, not a decimal point. Of course, I could easily have achieved this without using LINQ. Example 10-2 will produce the same results.

Example 10-2. The non-LINQ equivalent

```
CultureInfo[] allCultures = CultureInfo.GetCultures(CultureTypes.AllCultures);
foreach (CultureInfo culture in allCultures)
{
    if (culture.NumberFormat.NumberDecimalSeparator == ",")
    {
        Console.WriteLine(culture.Name);
    }
}
```

Both examples have 8 non-blank lines of code, although if you ignore lines that contain only braces, Example 10-2 contains just four, two fewer than Example 10-1. Then again, if we count statements the LINQ example has just three, compared to four in the loop-based example. So it's difficult to argue convincingly that either approach is simpler than the other.

However, Example 10-1 has at least one significant advantage: the code that decides which items to choose is well separated from the code that decides what to do with those

items. Example 10-2, intermingles these two concerns: the code that picks the objects is half outside and half inside the loop.

Another difference is that Example 10-1 has a more declarative style: it focuses on what we want, not how to get it. The query expression describes the items we'd like, without mandating that this be achieved in any particular way. For this very simple example, that doesn't matter much, but for more complex examples, and particularly when using a LINQ provider for database access, it can be very useful to allow the provider a free hand in deciding exactly how to perform the query.

There are three parts to the query in Example 10-1. It begins, as all query expressions are required to begin, with a `from` clause, which specifies the source of the query. In this case, the source is an array of type `CultureInfo[]`, returned by the `CultureInfo` class's `GetCultures` method. As well as defining the source for the query, the `from` clause specifies a name, which in this example is `culture`. This is called the *range variable*, and we can use it in the rest of the query to represent a single item from the source. Clauses can run many times—the `where` clause in Example 10-1 runs once for every item in the collection, so the range variable will have a different value each time. This is reminiscent of the iteration variable in a `foreach` loop. In fact the overall structure of the `from` clause is similar—we have the variable that will represent an item from a collection, then the `in` keyword, then the source for which that variable will represent individual items. Just as a `foreach` loop's iteration variable is only in scope inside the loop, the range variable `culture` is only meaningful inside this query expression.

> Although analogies with `foreach` can be helpful for understanding the intent of LINQ queries, you should not take this too literally. For example, not all providers directly execute the expressions in a query. Some LINQ providers convert query expressions into database queries, in which case the C# code in the various expressions inside the query does not run in any conventional sense. So although it is true to say that the range variable represents a single value from the source, it's not always true to say that that clauses will execute once for every item they process, with the range value taking that item's value. It happens to be true for Example 10-1 because it uses LINQ to Objects, but it's not true for all providers.

The second part of the query in Example 10-1 is a `where` clause. This is optional; conversely, you're also allowed several in one query. A `where` clause filters the results, and the one in this example states that I only want the `CultureInfo` objects with a `NumberFormat` that indicates that the decimal separator is a comma.

The final part of the query is a `select` clause, and all query expressions end either with one of these or a `group` clause. This determines the final output of the query. This example indicates that we want each `CultureInfo` object that was not filtered out by the query. The `foreach` loop in Example 10-1 that prints out the results of the query only uses the `Name` property, so I could have written a query that only extracted that. As Example 10-3 shows, if I do this, I also need to change the loop, because the resulting query now produces strings instead of `CultureInfo` objects.

Example 10-3. Extracting just one property in a query

```
IEnumerable<string> commaCultures =
    from culture in CultureInfo.GetCultures(CultureTypes.AllCultures)
    where culture.NumberFormat.NumberDecimalSeparator == ","
    select culture.Name;

foreach (string cultureName in commaCultures)
{
    Console.WriteLine(cultureName);
}
```

This raises a question: in general, what type do query expressions have? In Example 10-1, `commaCultures` is an `IEnumerable<CultureInfo>` and in Example 10-3, it's an `IEnumerable<string>`. The output item type is determined by the final clause of the query—the `select` or, in some cases, the `group` clause. However, not all query expression result in an `IEnumerable<T>`. It depends on which LINQ provider you use—I've ended up with `IEnumerable<T>` because I'm using LINQ to Objects.

> It's very common to use the `var` keyword when declaring variables that hold LINQ queries. This is necessary if a `select` clause produces instances of an anonymous type, because there is no way to write the name of the resulting query's type. Even if anonymous types are not involved, `var` is still widely used, and there are two reasons. One is just a matter of consistency: some people feel that because you have to use `var` for some LINQ queries, you should use it for all of them. A slightly better argument is that LINQ query types often have verbose and ugly names, and `var` results in less cluttered code. I have a slight preference for `var` here for the second reason, but will make the type explicit if I believe it makes the code easier to understand.

How did C# know that I wanted to use LINQ to Objects? It's because I used an array as the source in the `from` clause. More generally, LINQ to Objects will be used when you specify any `IEnumerable<T>` as the source, unless a more specialized provider is available. However, this doesn't really explain how C# discovers the existence of providers in the first place, and how it chooses between them. To understand that, you need to know what the compiler does with a query expression.

How Query Expressions Expand

The compiler converts all query expressions into one or more method calls. Once it has done that, the LINQ provider is selected through exactly the same mechanisms that C# uses for any other method call. The compiler does not have any built in concept of what constitutes a LINQ provider, so it relies on convention. Example 10-4 shows what the compiler does with my second query expression, the one in Example 10-3.

Example 10-4. The effect of a query expression

```
IEnumerable<string> commaCultures =
    CultureInfo.GetCultures(CultureTypes.AllCultures)
    .Where(culture => culture.NumberFormat.NumberDecimalSeparator == ",")
    .Select(culture => culture.Name);
```

The `Where` and `Select` methods are examples of LINQ operators. A LINQ operator is nothing more than a method that conforms to one of the standard patterns. I'll describe these patterns later in this chapter, in "Standard LINQ Operators".

The code in Example 10-4 is all one statement, and I'm chaining method calls together—I call the `Where` method on the return value of `GetCultures`, and I call the `Select` method on the return value of `Where`. The formatting looks a little peculiar, but it's too long to go on one line, and even though it's not terribly elegant, I prefer to put the `.` at the start of the line when splitting chained calls across multiple lines, because it makes it much easier to see that each new line continues from where the last one left off. Leaving the period at the end of the preceding line looks much neater, but also makes it much easier to misread the code.

The compiler has turned the `where` and `select` clauses' expressions into lambdas. Notice that the range variable ends up as an argument to each lambda. This is one example of why you should not take the analogy between query expressions and `foreach` loops too literally. Unlike a `foreach` iteration variable, the range variable does not exist as a single conventional variable. In the query, it is just an identifier that represents an item from the source, and in expanding the query into method calls, C# may end up creating multiple real variables for a single range variable, like it has with the arguments for the two separate lambdas here.

All query expressions boil down to this sort of thing—chained method calls with lambdas. Some are more complex than others. The expression in Example 10-1 ends up with a simpler structure despite looking almost identical to Example 10-3. Example 10-5 shows how it expands. It turns out that when a query's `select` clause just passes the range variable straight through, the compiler interprets that as meaning that we want to pass the results of the preceding clause straight through without further processing, so it doesn't add a call to `Select`. (There is one exception to this: if you write a query expression that contains nothing but a `from` and a `select` clause, it will generate a call to `Select` even if the `select` clause is trivial.)

Example 10-5. How trivial select clauses expand

```
IEnumerable<string> commaCultures =
    CultureInfo.GetCultures(CultureTypes.AllCultures)
    .Where(culture => culture.NumberFormat.NumberDecimalSeparator == ",");
```

The compiler has to work harder if you introduce multiple variables within the query's scope. You can do this is with a `let` clause. Example 10-6 performs the same job as Example 10-3, but I've introduced a new variable called `numFormat` to refer to the number format. This makes my `where` clause shorter and easier to read, and in a more complex query that needed to refer to that format object multiple times, this technique could remove a lot of clutter.

Example 10-6. Query with let clause

```
IEnumerable<string> commaCultures =
    from culture in CultureInfo.GetCultures(CultureTypes.AllCultures)
    let numFormat = culture.NumberFormat
    where numFormat.NumberDecimalSeparator == ","
    select culture.Name;
```

When you write a query that has more than just a single range variable, the compiler automatically generates a hidden class with a field for each of the variables so that it can make all the variables available at every stage. To get the same effect with ordinary method calls, we'd need to do something similar, and the easiest way to do that would be to introduce an anonymous type to contain them, as Example 10-7 shows.

Example 10-7. How multi-variable query expressions expand (approximately)

```
IEnumerable<string> commaCultures =
    CultureInfo.GetCultures(CultureTypes.AllCultures)
    .Select(culture => new { culture, numFormat = culture.NumberFormat })
    .Where(vars => vars.numFormat.NumberDecimalSeparator == ",")
    .Select(vars => vars.culture.Name);
```

No matter how simple or complex they get, query expressions are simply a specialized syntax for method calls. This suggests how we might go about writing a custom source for a query expression.

Supporting Query Expressions

Because the C# compiler just converts the various clauses of a query expression into method calls, we can write a type that participates in these expressions by defining some suitable methods. To illustrate that the C# compiler really doesn't care what these methods do, Example 10-8 shows a class that makes absolutely no sense, but which nonetheless keeps C# happy when used from a query expression. The compiler just mechanically converts a query expression into a series of method calls, so if suitable-looking methods exist, the code will compile successfully.

Example 10-8. Nonsensical Where and Select

```
public class SillyLinqProvider
{
    public SillyLinqProvider Where(Func<string, int> pred)
    {
        Console.WriteLine("Where invoked");
        return this;
    }

    public string Select<T>(Func<DateTime, T> map)
    {
        Console.WriteLine("Select invoked, with type argument " + typeof(T));
        return "This operator makes no sense";
    }
}
```

I can use an instance of this class as the source of a query expression. That's crazy because this class does not in any way represent a collection of data, but the compiler doesn't care. It just needs certain methods to be present, so if I write the code in Example 10-9, the compiler will be perfectly happy even though the code doesn't make any sense.

Example 10-9. A meaningless query

```
var q = from x in new SillyLinqProvider()
        where int.Parse(x)
        select x.Hour;
```

The compiler converts this into method calls in exactly the same way that it did with the more sensible query in Example 10-1. Example 10-10 shows the result. If you're paying close attention, you'll have noticed that my range variable actually changes type part way through—my `Where` method requires a delegate that takes a string, so in that first lambda, `x` is of type `string`. But my `Select` method requires its delegate to take a `DateTime`, so that's the type of `x` in that lambda. (And it's all ultimately irrelevant because my `Where` and `Select` methods don't use these lambdas.) Again, this is nonsense, but it shows how mechanically the C# compiler converts queries to method calls.

Example 10-10. How the compiler transforms the meaningless query

```
var q = new SillyLinqProvider().Where(x => int.Parse(x)).Select(x => x.Hour);
```

Obviously it's not useful to write code that makes no sense. The reason I'm showing you this is to demonstrate that the query expression syntax knows nothing about semantics—the compiler has no particular expectation of what any of the methods it invokes will do. All that it requires is that they accept lambdas as arguments, and return something other than `void`.

Clearly the real work is happening elsewhere. It's the LINQ providers themselves that make things happen. So now I'll show what we would need to write to make the queries I showed in the first couple of examples work if LINQ to Objects didn't exist.

You've seen how LINQ to Objects queries are transformed into code such as that shown in Example 10-4, but this isn't the whole story. The `where` clause becomes a call to the `Where` method, but we're calling it on an array of type `CultureInfo[]`, a type that does not in fact have a `Where` method. This only works because LINQ to Objects defines an extension method—as I showed in Chapter 3, it's possible to add new methods to existing types, and LINQ to Objects does that for the `IEnumerable<T>` type. To use these extension methods, you need a `using` directive for the `System.Linq` namespace. (The extension methods are all defined by a static class called `Enumerable`, by the way.) Visual Studio adds such a directive to each C# file, so these methods are available by default. If you were to remove that directive, the compiler would produce this error for the query expression for Example 10-1 or Example 10-3:

```
error CS1935: Could not find an implementation of the query pattern for source
type 'System.Globalization.CultureInfo[]'.  'Where' not found.  Are you missing
a reference to 'System.Core.dll' or a using directive for 'System.Linq'?
```

In general, that error message's suggestion would be helpful, but in this case, I want to write my own LINQ implementation. Example 10-11 does this, and I've shown the whole source file because extension methods are sensitive to use of namespaces and `using` directives. The contents of the `Main` method should look familiar—this is the code from Example 10-3, but this time, instead of using the LINQ to Objects provider, it will use the extension methods from my `CustomLinqProvider` class. (Normally you make extension methods available with a `using` directive, but because `CustomLinqProvider` is in the same namespace as the `Program` class, all of its extension methods are automatically available to `Main`.)

Although Example 10-11 behaves as intended, you should not take this as an example of how a LINQ provider normally executes its queries. This does illustrate how LINQ providers put themselves in the picture, but as I'll show later, there are some issues with how this code goes on to perform the query. Also, it's obviously not complete—there's more to LINQ than `Where` and `Select`.

Example 10-11. A custom LINQ provider for CultureInfo[]

```
using System;
using System.Globalization;

namespace CustomLinqExample
{
    public static class CustomLinqProvider
```

```
    {
        public static CultureInfo[] Where(this CultureInfo[] cultures,
                                          Predicate<CultureInfo> filter)
        {
            return Array.FindAll(cultures, filter);
        }

        public static T[] Select<T>(this CultureInfo[] cultures,
                                    Func<CultureInfo, T> map)
        {
            var result = new T[cultures.Length];
            for (int i = 0; i < cultures.Length; ++i)
            {
                result[i] = map(cultures[i]);
            }
            return result;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            var commaCultures =
              from culture in CultureInfo.GetCultures(CultureTypes.AllCultures)
              where culture.NumberFormat.NumberDecimalSeparator == ","
              select culture.Name;

            foreach (string cultureName in commaCultures)
            {
                Console.WriteLine(cultureName);
            }
        }
    }
}
```

As you're now well aware, the query expression in `Main` will first call `Where` on the source, and will then call `Select` on whatever `Where` returns. As before, the source is the return value of `GetCultures`, which is an array of type `CultureInfo[]`. That's the type for which `CustomLinqProvider` defines extension methods, so this will invoke `CustomLinqProvider.Where`. That uses the `Array` class's `FindAll` method to find all of the elements in the source array that match the predicate. The `Where` method passes its own argument straight through to `FindAll` as the predicate, and as you know, when the C# compiler calls `Where`, it passes a lambda based on the expression in the LINQ query's `where` clause. That predicate will match the cultures that use a comma as their decimal separator, so the `Where` clause returns an array of type `CultureInfo[]` that only contains those cultures.

Next the code that the compiler created for the query will call `Select` on the `CultureInfo[]` array returned by `Where`. Arrays don't have a `Select` method, so the extension method in `CustomLinqProvider` will be used. My `Select` method is generic, so the compiler will need to work out what the type argument should be, and it can infer this from the expression in the `select` clause. First, the compiler transforms it into a lambda: `culture => culture.Name`. Because this becomes the second argument for `Select`, the compiler knows that we require a `Func<CultureInfo,`

`T>`, so it knows that the `culture` parameter must be of type `CultureInfo`. This enables it to infer that `T` must be `string`, because the lambda returns `culture.Name`, and that `Name` property's type is `string`. So the compiler knows that it is invoking `CustomLinqProvider.Select<string>`. (The deduction I just described is not specific to query expressions here by the way. The type inference takes place after the query has been transformed into method calls. The compiler would have gone through exactly the same process if we had started with the code in Example 10-4.)

The `Select` method will now produce an array of type `string[]` (because `T` is `string` here). It populates that array by iterating through the elements in the incoming `CultureInfo[]`, passing each `CultureInfo` as the argument to the lambda that extracts the `Name` property. So we end up with an array of strings, containing the name of each of the cultures that uses a comma as its decimal separator.

That's a slightly more realistic example than my `SillyLinqProvider`, because this does now provide the expected behavior. However, although the query produces the same strings as it did when using the real LINQ to Objects provider, the mechanism by which it did so is somewhat different. My `CustomLinqProvider` performed each operation immediately—the `Where` and `Select` methods both returned fully populated arrays. LINQ to Objects does something quite different. In fact, so do most LINQ providers.

Deferred Evaluation

If LINQ to Objects worked in the same way as my custom provider in Example 10-11, it would not cope well with Example 10-12. This has a `Fibonacci` method that returns a never-ending sequence—it will keep providing numbers from the Fibonacci series for as long as code keeps asking for them. I have used the `IEnumerable<BigInteger>` returned by this method as the source for a query expression. As you can see, I've left the default `using` directive for `System.Linq` in place at the start, so I'm back to using LINQ to Objects here.

Example 10-12. Query with an infinite source sequence

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Numerics;

class Program
{
    static IEnumerable<BigInteger> Fibonacci()
    {
        BigInteger n1 = 1;
        BigInteger n2 = 1;
        yield return n1;
        while (true)
        {
            yield return n2;
            BigInteger t = n1 + n2;
            n1 = n2;
            n2 = t;
        }
    }
```

```
    static void Main(string[] args)
    {
        var evenFib = from n in Fibonacci()
                      where n % 2 == 0
                      select n;

        foreach (BigInteger n in evenFib)
        {
            Console.WriteLine(n);
        }
    }
}
```

This will use the `Where` extension method that LINQ to Objects provides for `IEnumerable<T>`. If that worked the same way as my `CustomLinqExtension` class's `Where` method for `CultureInfo[]`, this program would never make it as far as printing out a single number. My `Where` method did not return until it had filtered the whole of its input and produced a fully populated array as its output. If the LINQ to Objects `Where` method tried that with my infinite Fibonacci enumerator, it would never finish.

In fact, Example 10-12 works perfectly—it produces a steady stream of output consisting of the Fibonacci numbers that are divisible by 2. So it's not attempting to perform the filtering when we call `Where`. Instead, its `Where` method returns an `IEnumerable<T>` that filters items on demand. It won't try to fetch anything from the input sequence until something asks for a value, at which point it will start retrieve one value after another from the source until the filter delegate says that a match has been found. It then returns that and doesn't try to retrieve anything more from the source until it is asked for the next item. Example 10-13 shows how you could implement this behavior by taking advantage of C#'s `yield return` feature.

Example 10-13. A custom deferred Where operator

```
public static class CustomDeferredLinqProvider
{
    public static IEnumerable<T> Where<T>(this IEnumerable<T> src,
                                          Func<T, bool> filter)
    {
        foreach (T item in src)
        {
            if (filter(item))
            {
                yield return item;
            }
        }
    }
}
```

The real LINQ to Objects implementation of `Where` is somewhat more complex. It detects certain special cases such as arrays and lists, and it handles those in a way that is slightly more efficiently than the general-purpose implementation that it falls back to for other types. However, the principle is the same for `Where` and all of the other operators: these methods do not perform the specified work. Instead, they return objects which will perform the work on demand. It's only when you attempt to retrieve the results of a query that anything really happens. This is called *deferred evaluation*.

Deferred evaluation has the benefit of not doing work until you need it, and it makes it possible to work with infinite sequences. However, it also has disadvantages. You may need to be careful to avoid evaluating queries multiple times. Example 10-14 makes this mistake, causing it to do much more work than necessary. This loops through several different numbers, and prints each one out using the currency format of each of the cultures that uses a comma as a currency separator.

> If you run this, you may find that most of the lines this code prints will contain `?` characters, indicating that the console cannot display the most of the currency symbols. In fact it can, it just needs permission. By default, the Windows console uses an 8-bit code page for backwards compatibility reasons. If you run the command `chcp 65001` it will switch into a UTF-8 code page, enabling it to print any Unicode characters supported by your chosen console font. You might want to configure the console to use either Consolas or Lucida Console to take best advantage of that.

Example 10-14. Accidental reevaluation of a deferred query

```
var commaCultures =
    from culture in CultureInfo.GetCultures(CultureTypes.AllCultures)
    where culture.NumberFormat.NumberDecimalSeparator == ","
    select culture;

object[] numbers = { 1, 100, 100.2, 10000.2 };

foreach (object number in numbers)
{
    foreach (CultureInfo culture in commaCultures)
    {
        Console.WriteLine(string.Format(culture, "{0}: {1:c}",
                          culture.Name, number));
    }
}
```

The problem with this code is that even though the `commaCultures` variable is initialized outside of the number loop, we iterate through it for each number. And because LINQ to Objects uses deferred evaluation, that means that the actual work of running the query is redone every time round the outer loop. So instead of evaluating that `where` clause once for each culture (354 times on my system) it ends up running four times for each culture (1,416 times on my system) because the whole query is evaluated once for each of the four items in the `numbers` array. It's not a disaster—the code still works correctly. But if you do this in a program that runs on a heavily loaded server, it will harm your throughput.

If you know you will need to iterate through the results of a query multiple times, consider using either the `ToList` or `ToArray` extension methods provided by LINQ to Objects. These immediately evaluate the whole query once, producing an `IList<T>` or a `T[]` array respectively (so you shouldn't use these methods on infinite sequences, obviously). You can then iterate through that as many times as you like without incurring any further costs (beyond the minimal cost inherent in reading all the elements of an array or list).

LINQ, Generics, and IQueryable<T>

Most LINQ providers use generic types. Nothing enforces this, but it is very common. LINQ to Objects uses `IEnumerable<T>`. Several of the database providers use a type called `IQueryable<T>`. More broadly, the pattern is to have some generic type *`Source`*`<T>`, where *`Source`* represents some source of items, and `T` the type of an individual item. A source type with LINQ support makes operator methods available on *`Source`*`<T>` for any `T`, and those operators also typically return *`Source`*`<TResult>` where `TResult` may or may not be different than `T`.

`IQueryable<T>` is interesting because it is designed to be used by multiple providers. This interface, its base `IQueryable`, and the related `IQueryProvider` are shown in Example 10-15.

Example 10-15. IQueryable and IQueryable<T>

```
public interface IQueryable : IEnumerable
{
    Type ElementType { get; }
    Expression Expression { get; }
    IQueryProvider Provider { get; }
}

public interface IQueryable<out T> : IEnumerable<T>, IQueryable, IEnumerable
{
}

public interface IQueryProvider
{
    IQueryable CreateQuery(Expression expression);
    IQueryable<TElement> CreateQuery<TElement>(Expression expression);
    object Execute(Expression expression);
    TResult Execute<TResult>(Expression expression);
}
```

The most obvious feature of `IQueryable<T>` is that it adds no members to its bases. That's because it's designed to be used entirely via extension methods. The `System.Linq` namespace defines all of the standard LINQ operators for `IQueryable<T>` as extension methods provided by the `Queryable` class. However, all of these simply defer to the `Provider` property defined by the `IQueryable` base. So unlike LINQ to Objects, where the extension methods on `IEnumerable<T>` define the behavior, an `IQueryable<T>` implementation is able to decide how to handle queries because it gets to supply the `IQueryProvider` that does the real work.

However, all `IQueryable<T>`-based LINQ providers have one thing in common: they interpret the lambdas as expression objects, not delegates. Example 10-16 shows the declaration of the `Where` extension methods defined for `IEnumerable<T>` and `IQueryable<T>`. Compare the `predicate` parameters.

Example 10-16. Enumerable vs Queryable

```
public static class Enumerable
{
    public static IEnumerable<TSource> Where<TSource>(
        this IEnumerable<TSource> source,
```

```
        Func<TSource, bool> predicate)
    ...
}

public static class Queryable
{
    public static IQueryable<TSource> Where<TSource>(
        this IQueryable<TSource> source,
        Expression<Func<TSource, bool>> predicate)
    ...
}
```

The `Where` extension for `IEnumerable<T>` (LINQ to Objects) takes a `Func<TSource,bool>`, and as you saw in Chapter 9, this is a delegate type. But the `Where` extension method for `IQueryable<T>` (used by numerous LINQ providers) takes `Expression<Func<TSource,bool>>`, and as you also saw in Chapter 9, this causes the compiler to build an object model of the expression and pass that as the argument.

The usual reason for a LINQ provider to use `IQueryable<T>` is because it wants these expression trees. So when a provider uses that interface, it usually means that it's going to inspect your query and convert it into something else, such as a SQL query.

There are some other common generic types that crop up in LINQ. Some LINQ features guarantee to produce items in a certain order and some do not. More subtly, a handful of operators produce items in an order that depends upon the order of their input. This can be reflected in the types for which the operators are defined, and the types they return. LINQ to Objects defines `IOrderedEnumerable<T>` to represent ordered data, and there's a corresponding `IOrderedQueryable<T>` type for `IQueryable<T>`-based providers. (Providers that use their own types tend to do something similar—Parallel LINQ defines an `OrderedParallelQuery<T>` for example.) These interfaces derive from their unordered counterparts such as `IEnumerable<T>` and `IQueryable<T>`, so all the usual operators are available, but they make it possible to define operators or other methods that need to take the existing order of their input into account. For example, in the "Ordering" section later in this chapter, I'll show a LINQ operator called `ThenBy`, which is only available on sources that are already ordered.

When looking at LINQ to Objects, this ordered/unordered distinction may seem unnecessary, because `IEnumerable<T>` always produces items in some sort of order. But some providers do not necessary do things in any particular order, perhaps because they parallelize query execution, or because they get a database to execute the query for them, and databases reserve the right to meddle with the order in certain cases if it enables them to work more efficiently.

Standard LINQ Operators

In this section, I will describe the standard operators that LINQ providers can supply. Where applicable, I will also describe the query expression equivalent, although many operators do not have a corresponding query expression form. Some LINQ features are only available through explicit method invocation. This is even true with certain operators that can be used in query expressions, because most operators are overloaded, and query expressions can't use some of the more advanced overloads.

> LINQ operators are not operators in the usual C# sense—they are not symbols such as `+` or `&&`. LINQ has its own terminology, and for this chapter, an operator is a query capability offered by a LINQ provider. In C# it looks like a method.

All of these operators have something in common: they have all been designed to support composition. Operators not only take some type representing a set of items (e.g, an `IEnumerable<T>`) most of them also return something representing a set of items. The item type is not always the same—in some cases an operator might take some `IEnumerable<T>` as input, and produce `IEnumerable<TResult>` as output, where `TResult` does not have to be the same as `T`. Even so, you can still chain the things together in any number of different ways. Part of the reason this works is that LINQ operators are like mathematical functions, in that they do not modify their inputs—they produce a new result that is based on their operands. (Functional programming languages typically have the same characteristic.) This means that not only are you free to plug operators together in arbitrary combinations without fear of side effects, you are also free to use the same source as the input to multiple queries, because no LINQ query will ever modify its input. Each operator returns a new query based on its input.

> It is possible to write an `IEnumerable<T>` implementation in which iterating through the items has side effects. However, this is a bad idea, particularly if you are using LINQ, because LINQ is designed around the assumption that you can enumerate a collection without consequences other than consuming resources such as CPU time.

Nothing enforces this functional style. As you saw with my `SillyLinqProvider`, the compiler doesn't care what a method representing a LINQ operator does. However, the convention is that operators are functional, in order to support composition. The built-in LINQ providers all work this way.

Not all providers provide complete support for all operators. The main providers in the .NET Framework such as LINQ to Objects, Entities, or SQL are as comprehensive as they can be, but I will show that there are some situations in which certain operators will not make sense.

To demonstrate the operators in action, I need some source data. Many of the examples in the following sections will use the code in Example 10-17.

Example 10-17. Sample input data for LINQ queries

```
public class Course
{
    public string Title { get; set; }

    public string Category { get; set; }

    public int Number { get; set; }

    public DateTime PublicationDate { get; set; }

    public TimeSpan Duration { get; set; }

    public static readonly Course[] Catalog =
    {
```

```
        new Course
        {
            Title = "Elements of Geometry",
            Category = "MAT", Number = 101, Duration = TimeSpan.FromHours(3),
            PublicationDate = new DateTime(2009, 5, 20)
        },
        new Course
        {
            Title = "Squaring the Circle",
            Category = "MAT", Number = 102, Duration = TimeSpan.FromHours(7),
            PublicationDate = new DateTime(2009, 4, 1)
        },
        new Course
        {
            Title = "Recreational Organ Transplantation",
            Category = "BIO", Number = 305, Duration = TimeSpan.FromHours(4),
            PublicationDate = new DateTime(2002, 7, 19)
        },
        new Course
        {
            Title = "Hyperbolic Geometry",
            Category = "MAT", Number = 207, Duration = TimeSpan.FromHours(5),
            PublicationDate = new DateTime(2007, 10, 5)
        },
        new Course
        {
            Title = "Oversimplified Data Structures for Demos",
            Category = "CSE", Number = 104, Duration = TimeSpan.FromHours(2),
            PublicationDate = new DateTime(2012, 2, 21)
        },
        new Course
        {
            Title = "Introduction to Human Anatomy and Physiology",
            Category = "BIO", Number = 201, Duration = TimeSpan.FromHours(12),
            PublicationDate = new DateTime(2001, 4, 11)
        },
    };
}
```

Filtering

One of the simplest operators is `Where`, which filters its input. You provide a function that takes an individual item and returns a `bool`, and `Where` returns an object representing the items from the input for which the predicate is true. (Conceptually this is very similar to the `FindAll` method available on `List<T>` and array types, but using deferred execution.)

As you've already seen, query expressions represent this with a `where` clause. However, an overload of the `Where` operator provides an additional feature that is not accessible from a query expression. You can write a filter lambda that takes two arguments: an item from the input and an index representing that item's position in the source. Example 10-18 uses this form to remove every 2nd number from the input, and it also removes courses shorter than three hours.

Example 10-18. Where operator with index

```
IEnumerable<Course> q = Course.Catalog.Where(
    (course, index) => (index % 2 == 0) && course.Duration.TotalHours >= 3);
```

Indexed filtering is only meaningful for ordered data. It always works with LINQ to Objects, because that works with `IEnumerable<T>`, which produces items one after another, but not all LINQ providers process items in sequence. For example, if you're using LINQ to Entities, the LINQ queries you write in C# will be handled on the database. Unless a query explicitly requests some particular order, a database is usually free to process items in whatever order it sees fit, possibly processing items in parallel. In some cases, a database may have optimization strategies that enable it to produce the results a query requires with a process that bears little resemblance to the original query, so it might not even be meaningful talk about, say, the 14th item handled by a `WHERE` clause. Consequently, it you were to write a query similar to Example 10-18 using LINQ to Entities, executing the query would cause an exception, complaining that the indexed `Where` operator is not applicable. If you're wondering why the overload is even available from a provider that doesn't support it, it's because LINQ to Entities uses `IQueryable<T>`, so all the standard operators are available at compile time; providers that choose to use `IQueryable<T>` can only report the non-availability of operators at runtime.

Even so, you might have expected the exception to emerge when you invoke `Where`, instead of when you try to execute the query (i.e., when you first try to retrieve one or more items). However, providers that convert LINQ queries into some other form such as a SQL query typically defer all validation until you execute the query. This is because some operators may be valid only in certain scenarios, meaning that the provider may not know whether any particular operator will work until you've finished building the whole query. It would be inconsistent if errors caused by non-viable queries sometimes emerged while building the query, and sometimes when executing it, so even in cases where a provider can know for certain that a particular operator will fail, it will usually wait until you execute the query to tell you.

The `Where` operator's filter lambda must take an argument of the item type (the `T` in `IEnumerable<T>`, for example) and it must return a `bool`. You may remember from Chapter 9 that the class library defines a suitable delegate type called `Predicate<T>`, but I also mentioned in that chapter that LINQ avoids this, and it should now be clear why. The indexed version of the `Where` operator cannot use `Predicate<T>` because there's an additional argument, so that overload uses `Func<T, int, bool>`. LINQ providers tend to use `Func` across the board so that operators with similar meanings have similar-looking signatures, so although the unindexed form of `Where` could use `Predicate<T>`, most providers use `Func<T, bool>` instead, to be consistent with the indexed version. (C# doesn't care which you use—query expressions still work if the provider uses `Predicate<T>`, as my custom `Where` operator in Example 10-11 shows, but none of Microsoft's providers do this.)

LINQ defines another filtering operator: `OfType<T>`. This is useful if your source contains mixture of different item types—perhaps the source is an `IEnumerable<object>` and you'd like to filter this down to only the items of type `string`. Example 10-19 shows how the `OfType<T>` operator can produce just those objects that are strings.

Example 10-19. The OfType<T> operator

```
static void ShowAllStrings(IEnumerable<object> src)
```

```
{
    var strings = src.OfType<string>();
    foreach (string s in strings)
    {
        Console.WriteLine(strings);
    }
}
```

Both `Where` and `OfType<T>` will produce empty sequences if none of the objects in the source meet the requirements. This is not considered to be an error—empty sequences are quite normal in LINQ. Many operators can produce them as output, and most operators can cope with them as input.

Select

When writing a query, we may want to extract only certain pieces of data from the source items. I've shown examples that have displayed only the title of a course, or its category, even though the query returned a sequence in which each item was the entire course object. The `select` clause at the end of most queries lets us supply a lambda that will be used to produce the final output items, and there are a couple of reasons we might want to make our `select` clause do more than simply passing each item straight through. We might want to pick just one specific piece of information from each item, or we might want to transform it into something else entirely.

You've seen several `select` clauses already, and I showed in Example 10-3 that the compiler turns them into a call to `Select`. However, as with many LINQ operators, the version accessible through a query expression is not the only option. There's one other overload, which provides not just the input item from which to generate the output item, but also the index of that item. Example 10-20 uses this to generate a numbered list of course titles.

Example 10-20. Select operator with index

```
IEnumerable<string> nonIntro = Course.Catalog.Select((course, index) =>
      string.Format("Course {0}: {1}", index + 1, course.Title));
```

Be aware that the zero-based index passed into the lambda will be based on what comes into the `Select` operator, and will not necessarily represent the item's original position in the source. This might not produce the results you were hoping for in code such as Example 10-21.

Example 10-21. Indexed Select downstream of Where operator

```
IEnumerable<string> nonIntro = Course.Catalog
    .Where(c => c.Number >= 200)
    .Select((course, index) => string.Format("Course {0}: {1}",
                                             index, course.Title));
```

This code will select the courses found at indexes 2, 3, and 5 respectively in the `Course.Catalog` array, because those are the courses whose `Number` property satisfies the `Where` expression. However, this query will number the three courses as 0, 1, and 2, because the `Select` operator only sees the items the `Where` clause let through. As far as it is concerned, there are only three items because the `Select` clause never had access to the original source. If you wanted the indices relative to the original collection, you'd need to extract those upstream of the `Where` clause, as Example 10-22 shows.

Example 10-22. Indexed Select upstream of Where operator

```
IEnumerable<string> nonIntro = Course.Catalog
    .Select((course, index) => new { course, index })
    .Where(vars => vars.course.Number >= 200)
    .Select(vars => string.Format("Course {0}: {1}",
                                  vars.index, vars.course.Title));
```

The indexed `Select` operator is similar to the indexed `Where` operator. So as you would probably expect, not all LINQ providers support it in all scenarios.

Data shaping and anonymous types

If you are using a LINQ provider to access a database, the `Select` operator can offer an opportunity to reduce the quantity of data you fetch from the database, which could reduce the load on the server. When you use a data access technology such as the Entity Framework or LINQ to SQL to execute a query that returns a set of objects representing persistent entities, there's a tradeoff between doing too much work up front, and having to do lots of extra deferred work. Should those frameworks fully populate all of the object properties that correspond to columns in various database tables? Should they also load related objects? In general, it's more efficient not to fetch data you're not going to use, and data that is not fetched up front can always be loaded later on demand. However, if you try to be too frugal in your initial request, you may ultimately end up making a lot of extra requests to fill in the gaps, which could outweigh any benefit from avoiding unnecessary work.

When it comes to related entities, the Entity Framework and LINQ to SQL allow you to configure what should be prefetched and what should be loaded on demand, but for any particular entity that gets fetched, all properties relating to columns are typically fully populated. This means queries that request whole entities end up fetching all the columns for any row that they touch.

If you only needed to use one or two columns, this is relatively expensive. Example 10-23 uses this somewhat inefficient approach. (This is based on one of Microsoft's example databases, the AdventureWorks Light 2008 sample.) It shows a fairly typical LINQ to Entities query.

Example 10-23. Fetching more data than is needed

```
var pq = from product in dbCtx.Products
           where product.ListPrice > 3000
           select product;
foreach (var prod in pq)
{
    Console.WriteLine("{0} ({2}): {1}", prod.Name, prod.ListPrice, prod.Size);
}
```

This LINQ provider translates the `where` clause into an efficient SQL equivalent. However, the SQL `SELECT` clause retrieves all 17 columns from the table. Compare that with Example 10-24. This modifies only one part of the query: the LINQ `select` clause now returns an instance of an anonymous type that contains only those properties we require. (The loop that follows the query can remain the same. It uses `var` for its iteration variable, which will work fine with the anonymous type, which provides the three properties that loop requires.)

Example 10-24. A select clause with an anonymous type

```
var pq = from product in dbCtx.Products
            where (product.ListPrice > 3000)
            select new { product.Name, product.ListPrice, product.Size };
```

The code produces exactly the same results, but it generates a much more compact SQL query that requests only the `Name`, `ListPrice`, and `Size` columns. This will produce a significantly smaller response because it's no longer dominated by data we don't need, reducing the load on the network connection to the database server, and also resulting in faster processing because the data will take less time to arrive. This technique is called *data shaping*.

This approach will not always be an improvement. For one thing, it means you are working directly with data in the database instead of using entity objects. This might mean working at a lower level of abstraction than would be possible if you use the entity types, which might increase development costs. Also, in some environments, database administrators do not allow ad hoc queries, in which case you won't have the flexibility to use this technique.

Projecting the results of a query into an anonymous type is not limited to database queries by the way. You are free to do this with any LINQ provider, such as LINQ to Objects. It can sometimes be a useful way to get structured information out of a query without needing to define a class specially.

Projection and mapping

The `Select` operator is sometimes referred to as *projection*, and it is the same operation that many languages call *map*, which provides a slightly different way to think about the `Select` operator. So far, I've presented `Select` as a way to choose what comes out of a query, but another way to look at it is as a way to apply a transformation to every object in the source. Example 10-25 uses `Select` to produce modified versions of a list of numbers. It variously doubles the numbers, squares them, and turns them into text.

Example 10-25. Using Select to transform numbers

```
int[] numbers = { 0, 1, 2, 3, 4, 5 };

IEnumerable<int> doubled = numbers.Select(x => 2 * x);
IEnumerable<int> squared = numbers.Select(x => x * x);
IEnumerable<string> numberText = numbers.Select(x => x.ToString());
```

Incidentally, `Select` is conceptually the same operation as one part of what Google calls Map Reduce. (LINQ's name for *reduce* is `Aggregate`.) Of course, the interesting thing about Map Reduce is not the map or reduce operations—they are pretty ordinary—it's the highly parallelized distributed execution. Microsoft Research developed a distributed version of LINQ called DryadLINQ. This was being developed into a product called LINQ to HPC (High Performance Computing), but that was sadly abandoned near the end of its beta cycle. However, there is some scope for parallelization: one of the providers that ships with .NET is Parallel LINQ, which I'll discuss later.

SelectMany

The `SelectMany` LINQ operator is used in query expressions that have multiple `from` clauses. It's called `SelectMany` because instead of selecting a single output item for each input item, you provide it with a lambda that produces a whole collection for each input item. The resulting query produces all of the objects from all of these collections, as

though each of the collections your lambda returns were merged into one. There are a couple of ways of thinking about this operator. One is that is provides a means of flattening two levels of hierarchy—a collection of collections—into a single level. But another way to look at it is as a Cartesian product, that is, a way to produce every possible combination from some input sets.

Example 10-26 shows how to use this operator in a query expression, and Example 10-27 shows the equivalent of that query expression, using the operator directly. This code highlights the Cartesian-product-like behavior. It prints every combination of the letters A, B, and C with a single digit from 1 to 5, i.e. A1, B1, C1, A2, B2, C2, etc.

Example 10-26. Using SelectMany from a query expression

```
int[] numbers = { 1, 2, 3, 4, 5 };
string[] letters = { "A", "B", "C" };

IEnumerable<string> combined = from number in numbers
                               from letter in letters
                               select letter + number;
foreach (string s in combined)
{
    Console.WriteLine(s);
}
```

Example 10-27. SelectMany operator

```
IEnumerable<string> combined = numbers.SelectMany(
        number => letters,
        (number, letter) => letter + number);
```

This example uses two fixed collections—the second `from` clause returns the same `letters` collection every time. However, you can make the expression in the second `from` clause return a value based on the current item from the first `from` clause. You can see in Example 10-27 that the first lambda passed to `SelectMany` (which actually corresponds to the second `from` clause's final expression) receives the current item from the first collection through its `number` argument, so you can use that to choose a different collection for each item from the first collection. I can use this to exploit `SelectMany`'s flattening behavior.

I've copied a jagged array from an example in Chapter 5 into Example 10-28, which then processes it with a query containing two `from` clauses. Note that the expression in the second `from` clause is now `item`, the range variable of the first `from` clause.

Example 10-28. Flattening a jagged array

```
int[][] arrays =
{
    new[] { 1, 2 },
    new[] { 1, 2, 3, 4, 5, 6 },
    new[] { 1, 2, 4 },
    new[] { 1 },
    new[] { 1, 2, 3, 4, 5 }
};

IEnumerable<int> combined = from item in arrays
                            from number in item
                            select number;
```

The first `from` clause asks to iterate over each item in the top-level array. That item is also an array of course, and the second `from` clause asks to iterate over each of these nested arrays. This nested array's type is `int[]`, so the range variable of the second `from` clause, `number`, represents an `int` from that nested array. The `select` clause just returns each of these `int` values.

The resulting sequence provides every number in the arrays in turn. It has flattened the jagged array into a simple linear sequence of numbers. This behavior is conceptually similar to writing a nested pair of loops, one iterating over the outer `int[][]` array, and an inner loop iterating over the contents of each individual `int[]` array.

The compiler uses the same overload of `SelectMany` for Example 10-28 as it does for Example 10-27, but there's an alternative in this case. The final `select` clause is simpler in Example 10-28—it just passes on items from the second collection unmodified, which means the simpler overload shown in Example 10-29 does the job equally well. With this overload, we just provide a single lambda, which chooses the collection that `SelectMany` will expand for each of the items in the input collection.

Example 10-29. SelectMany without item projection

```
var combined = arrays.SelectMany(item => item);
```

That's a somewhat terse bit of code, so in case it's not clear how quite how that could end up flattening the array, Example 10-30 shows how you might implement `SelectMany` for `IEnumerable<T>` if you had to write it yourself.

Example 10-30. One implementation of SelectMany

```
static IEnumerable<T2> MySelectMany<T, T2>(
            this IEnumerable<T> src, Func<T, IEnumerable<T2>> getInner)
{
    foreach (T itemFromOuterCollection in src)
    {
        IEnumerable<T2> innerCollection = getInner(itemFromOuterCollection);
        foreach (T2 itemFromInnerCollection in innerCollection)
        {
            yield return itemFromInnerCollection;
        }
    }
}
```

Why does the compiler not use the simpler option shown in Example 10-29? The C# language specification defines how query expressions are translated into method calls, and it only mentions the overload shown in Example 10-26. Perhaps the reason the specification doesn't mention the simpler overload is to reduce the demands C# makes of types that want to support this double-`from` query form—you'd only need to write one method to enable this syntax for your own types. However, .NET's various LINQ providers are more generous, providing this simpler overload for the benefit of developers who choose to use the operators directly. In fact most providers define two more overloads: there are versions of both the `SelectMany` forms we've seen so far that also pass an item index to the first lambda. (The usual caveats about indexed operators apply, of course.)

Although Example 10-30 gives a reasonable idea of what LINQ to Objects does in `SelectMany`, it's not the exact implementation. There are optimizations for special cases. Moreover, other providers may use very different strategies. Databases often have

built-in support for Cartesian products, so some providers may implement `SelectMany` in terms of that.

Ordering

In general, LINQ queries do not guarantee to produce items in any particular order unless you explicitly define the order you require. You can do this in a query expression with an `orderby` clause. As Example 10-31 shows, you specify the expression by which you'd like the items to be ordered, and a direction—so this will produce a collection of courses ordered by ascending publication date. As it happens, `ascending` is the default, so can leave off that qualifier without changing the meaning. As you've probably guessed, you can specify `descending` to reverse the order.

Example 10-31. Query expression with orderby clause

```
var q = from course in Course.Catalog
        orderby course.PublicationDate ascending
        select course;
```

The compiler transforms the `orderby` clause in Example 10-31 to a call to the `OrderBy` method, and it would use `OrderByDescending` if you had specified a `descending` sort order. With source types that make a distinction between ordered and unordered items, these operators return the ordered type, e.g. `IOrderedEnumerable<T>` for LINQ to Objects, and `IOrderedQueryable<T>` for `IQueryable<T>`-based providers.

> With LINQ to Objects, these operators have to retrieve every element from their input before they can produce any output elements. An ascending `OrderBy` can only know which item to return first once it has found the lowest item, and it won't know for certain which is the lowest until it has seen all of them. Some providers will have additional knowledge about the data that can enable more efficient strategies. (E.g., a database may be able to use an index to return values in the order required.)

The `OrderBy` and `OrderByDescending` operators each have two overloads, only one of which is available from a query expression. If you invoke the methods directly, you can supply an additional parameter of type `IComparer<TKey>`, where `TKey` is the type of the expression by which the items are being sorted. This is likely to be important if you sort based on a `string` property, because there are several different orderings for text, and you may need to choose one based on your application's locale, or you may want to specify a culture-invariant ordering.

The expression that determines the order in Example 10-31 is very simple—it just retrieves the `PublicationDate` property from the source item. You can write more complex expressions if you want to. If you're using a provider that translates a LINQ query into something else, there may be limitations. If the query runs on the database, you may be able to refer to other tables—the provider might be able to convert an expression such as `product.ProductCategory.Name` into a suitable join. However, you will not be able to run any old code in that expression, because it has to be something that the database can execute. But LINQ to Objects just invokes the expression once for each object, so you really can put whatever code you like in there.

You may want to sort by multiple criteria. You should **not** do this by writing multiple `orderby` clauses. Example 10-32 makes this mistake.

Example 10-32. How NOT to apply multiple ordering criteria

```
var q = from course in Course.Catalog
        orderby course.PublicationDate ascending
        orderby course.Duration descending // BAD! Could discard previous order
        select course;
```

This code orders the items by publication date, and then by duration, but does so as two separate and unrelated steps. The second `orderby` clause only guarantees that the results will be in the order specified in that clause, and does not guarantee to preserve anything about the order in which the elements came in. If what you actually wanted was for the items to be in order of publication date, and for any items with the same publication date to be ordered by descending duration, you would need to write the query in Example 10-33.

Example 10-33. Multiple ordering criteria in a query expression

```
var q = from course in Course.Catalog
        orderby course.PublicationDate ascending, course.Duration descending
        select course;
```

LINQ defines separate operators for this multi-level ordering: `ThenBy` and `ThenByDescending`. Example 10-34 shows how to achieve the same effect as the query expression in Example 10-33 by invoking the LINQ operators directly. For LINQ providers whose types make a distinction between ordered and unordered collections, these two operators will only be available on the ordered form, such as `IOrderedQueryable<T>` or `IOrderedEnumerable<T>`. If you were to try to invoke `ThenBy` directly on `Course.Catalog`, you would get a compiler error.

Example 10-34. Multiple ordering criteria with LINQ operators

```
var q = Course.Catalog
    .OrderBy(course => course.PublicationDate)
    .ThenByDescending(course => course.Duration);
```

You will find that some LINQ operators preserve some aspects of ordering even if you do not ask them to. For example, LINQ to Objects will typically produce items in the same order in which they appeared in the input unless you write a query that causes it to change the order. But this is simply an artifact of how LINQ to Objects works, and you should not rely on it in general. In fact even when you are using that particular LINQ provider, you should check with the documentation to see whether the order you're getting is guaranteed, or just an accident of implementation. In general, if you care about the order, you should always write a query that makes that explicit.

Containment Tests

LINQ defines various standard operators for discovering things about what the collection contains. Some providers may be able to implement these operators without needing to inspect every item. (For example, a database-based provider may use a `WHERE` clause, and the database may be able to use an index to evaluate that without needing to look at every element.) However, there are no restrictions—you can use these operators however you like, and it's up to the provider to discover whether it can exploit a shortcut.

> Unlike most LINQ operators, these return neither a collection nor an item from their input. They just return true or false, or in some cases, a count.

The simplest operator is `Contains`. There are two overloads: one takes an item, while the other takes an item and an `IEqualityComparer<T>` so that you can customize how the operator determines whether an item in the source is the same as the specified item. `Contains` returns true if the source contains the specified item and false if it does not. (If you use the single-argument version with a collection that implements `IList<T>`, LINQ to Objects will detect that, and its implementation of `Contains` just defers to the collection. If you either use a non-`IList<T>` collection, or you provide a custom equality comparer, it has to examine every item in the collection.)

If instead of looking for a particular value, you want to know whether a collection contains any values that satisfy some particular criteria, you can use the `Any` operator. This takes a predicate, and it returns true if the predicate is true for at least one item in the source. If you want to know how many items match some criteria, you can use the `Count` operator. This also takes a predicate, and instead of returning a `bool`, it returns an `int`. If you are working with very large collections, the range of `int` may be insufficient, in which case you can use the `LongCount` operator, which returns a 64-bit count. (This is likely to be overkill for most LINQ to Objects applications, but it could matter when the collection lives in a database.)

The `Any`, `Count`, and `LongCount` operators have overloads that do not take any arguments. For `Any`, this tells you whether the source contains at least one element, and for `Count` and `LongCount`, these overloads tell you how many elements the source contains.

> Be wary of code such as `if (q.Count() > 0)`. Calculating the exact count may require the entire query to be evaluated, and in any case, it is likely to require more work than simply answering the question: is this empty? If `q` refers to a LINQ query, writing `if (q.Any())` is likely to be more efficient. (This is not necessary for list-like collections, where retrieving an element count is cheap, and may actually be more efficient than the `Any` operator.)

A close relative to the `Any` operator is the `All` operator. This one is not overloaded—it takes a predicate, and it returns true if and only if the source contains no items that do not match the predicate. I used an awkward double negative in the preceding sentence for a reason: `All` returns true when applied to an empty sequence, because an empty sequence certainly doesn't contain any elements that fail to match the predicate for the simple reason that it doesn't contain any elements at all.

[This next paragraph contains some characters with slightly out-there Unicode code points. It is brought to you by the quantifiers ∃ and ∀ (codepoints 2203 and 2200), the

Element Of character ∈ (codepoint 2208), the Logical And character ∧ (codepoint 2227)

and the Mathematical Double-Struck Capital, 𝕍 (codepoint 1D54D). That last one is

especially awesome because it's not even in the basic multilingual plane. Word has chosen the Cambria Math font to make this work, but presumably we need to be careful that these don't get lost when this goes to production.]

This may seems like a curiously pig-headed form of logic. It's reminiscent of the child who, when asked "Have you eaten your vegetables?" unhelpfully replies "I ate all the vegetables I put on my plate," neglecting to mention that he didn't put any vegetables on his plate in the first place. It's not technically untrue, but it fails to provide the information the parent was looking for. Nonetheless, the operators work this way for a reason: they correspond to some standard mathematical logical operators. `Any` is the

existential quantifier, usually written as a backwards E (∃) and pronounced "there exists"

and `All` is the *universal quantifier*, usually written as an upside-down A (∀) and

pronounced "for all". Mathematicians long ago agreed on a convention for statements

that apply the universal quantifier to an empty set. For example, defining 𝕍 as the set of

all vegetables I can assert that ∀{v : (v ∈ 𝕍) ∧ putOnPlateByMe(v)} eatenByMe(v), or in

English, for each vegetable that I put on my plate, it is true to say that I ate that vegetable. This statement is deemed to be true if the set is empty, and rather pleasingly, the proper term for such a statement is a *vacuous truth*. Perhaps mathematicians don't like vegetables either.

Specific Items and Subranges

It can be useful to write a query that produces just a single item. Perhaps you're looking for the first object in a list that meets certain criteria, or maybe you want to fetch information in a database identified by a particular key. LINQ defines several operators

that can do this, and some related ones for working with a subrange of the items a query might return.

Use the `Single` operator when you have a query that you believe should produce exactly one result. Example 10-35 shows just such a query—it looks up a course by its category and number, and in my sample data, this uniquely identifies a course.

Example 10-35. Appling the Single operator to a query

```
var q = from course in Course.Catalog
        where course.Category == "MAT" && course.Number == 101
        select course;

Course geometry = q.Single();
```

Because LINQ queries are built by chaining operators together, we can take the query built by the query expression and add on another operator, the `Single` operator in this case. While most operators would return an object representing another query—an `IEnumerable<T>` here since we're using LINQ to Objects—`Single` is different. Like `ToArray` and `ToList`, the `Single` operator evaluates the query immediately, and it then returns the one and only object that the query produced. If the query fails to produce exactly one object—perhaps it produces no items, or two—this will throw an `InvalidOperationException`.

There's an overload of the `Single` operator that takes a predicate. As Example 10-36 shows, this allows us to express the same logic as the whole of Example 10-35 more compactly. (As with the `Where` operator, all the predicate-based operators in this section use `Func<T,bool>`, not `Predicate<T>`.)

Example 10-36. Single operator with predicate

```
Course geometry = Course.Catalog.Single(
    course => course.Category == "MAT" && course.Number == 101);
```

The `Single` operator is unforgiving: if your query does not return exactly one item, it will fail. There's a slightly more flexible variant called `SingleOrDefault` which allows a query to return either one item or no items. If the query returns nothing, this method returns the default value for the item type, i.e. null if it's a reference type, zero if it's a numeric type, and false if the type is `bool`. Multiple matches still cause an exception. As with `Single`, there are two overloads, one with no arguments for use on a source that you believe contains no more than one object, and one that takes a predicate lambda.

LINQ defines two related operators, `First` and `FirstOrDefault`, each of which offer overloads taking no arguments or a predicate. For sequences containing zero or one items, these behave in exactly the same way as `Single` and `SingleOrDefault`: they return the item if there is one, and if there isn't, `First` will throw an exception while `FirstOrDefault` will return `null` or an equivalent value. However, these operators respond differently when there are multiple results—instead of throwing an exception, they just pick the first result and return that, discarding the rest. This might be useful if you want to find the most expensive item in a list—you could order a query by descending price and then pick the first result. Example 10-37 uses a similar technique to pick the longest course from my sample data.

Example 10-37. Using First to select the longest course

```
var q = from course in Course.Catalog
        orderby course.Duration descending
        select course;
Course longest = q.First();
```

If you have a query that doesn't guarantee any particular order for its results, these operators will pick one item arbitrarily.

> Do not use `First` or `FirstOrDefault` unless you expect there to be multiple matches, and you only want to process one of them. Some developers use these when they expect only a single match. The operators will work of course, but the `Single` and `SingleOrDefault` operators more accurately express your expectations. They will let you know when your expectations were misplaced by throwing an exception when there are multiple matches. If your code embodies incorrect assumptions, it's usually best to know about it instead of plowing on regardless.

The existence of `First` and `FirstOrDefault` raises an obvious question: can I pick the last item? And yes, there are also `Last` and `LastOrDefault` operators, and again, each offers two overloads, one taking no arguments, and one taking a predicate.

The next obvious question is: what if I want a particular element that's neither the first nor the last. Your wish is, in this particular instance, LINQ's command, because it offers `ElementAt` and `ElementAtOrDefault` operators, both of which take just an index. (There are no overloads.) This provides a way to access elements of any `IEnumerable<T>` by index, but be careful: if you ask for the 10,000th element, these operators may need to request and discard the first 9,999 elements to get there. As it happens, LINQ to Objects detects when the source object implements `IList<T>`, in which case it uses the indexer to retrieve the element directly instead of going the slow way round. But not all `IEnumerable<T>` implementations support random access, so these operators can be very slow. In particular, even if your source implements `IList<T>`, once you've applied one or more LINQ operators to it, the output of those operators will typically not support indexing. So it would be particularly disastrous to use `ElementAt` in a loop of the kind shown in Example 10-38.

Example 10-38. How NOT to use ElementAt

```
var mathsCourses = Course.Catalog.Where(c => c.Category == "MAT");
for (int i = 0; i < mathsCourses.Count(); ++i)
{
    // Never do this!
    Course c = mathsCourses.ElementAt(i);
    Console.WriteLine(c.Title);
}
```

Even though `Course.Catalog` is an array, I've filtered its contents with the `Where` operator, which returns a query of type `IEnumerable<Course>` that does not implement `IList<Course>`. The first iteration won't be too bad—I'll be passing `ElementAt` an index of 0, so it just returns the first match, and with my sample data, the very first item `Where` inspects will match. But the second time around the loop, we're calling `ElementAt` again. The query that `mathsCourses` refers to does not keep track of where we got to in the previous loop—it's an `IEnumerable<T>`, not an

`IEnumerator<T>`—so this will start again. `ElementAt` will ask that query for the first item, which it will promptly discard, and then it will ask for the next item, and that becomes the return value. So the `Where` query has now been executed twice—the first time, `ElementAt` only asked it for one item, and then the second time it asked it for two, so it has processed the first course twice now. The third time round the loop (which happens to be the final time) we do it all again but this time, `ElementAt` will discard the first two matches and will return the third, so now it has looked at the first course three times, the second one twice, and the third and fourth courses once. (The third course in my sample data is not in the `MAT` category, so the `Where` query will skip over this when asked for the third item.) So to retrieve three items, I've evaluated the `Where` query three times, causing it to evaluate my filter lambda seven times.

In fact it's worse than that, because the `for` loop will also invoke that `Count` method each time, and with a non-indexable source such as the one returned by `Where`, `Count` has to evaluate the entire sequence—the only way the `Where` operator can tell you how many items match is to look at all of them. So this code fully evaluates the query returned by `Where` three times in addition to the three partial evaluations performed by `ElementAt`. We get away with it here because the collection is small, but if I had an array with 1,000 elements, all of which turned out to match the filter, we'd be fully evaluating the `Where` query 1,000 times, and performing partial evaluations another 1,000 times. Each full evaluation calls the filter predicate 1,000 times, and the partial evaluations here will do so on average 500 times, so the code would end up executing the filter 1,500,000 times. Iterating through the `Where` query with `foreach` loop would evaluate the query just once, executing the filter expression 1,000 times, and would produce the same results.

So be careful with both `Count` and `ElementAt`. If you use them in a loop that iterates over the collection on which you invoke them, the resulting code will have $O(n^2)$ complexity.

All of the operators I've just described return a single item from the source. There are two more operators that also get selective about which items to use, but which can return multiple items: `Skip` and `Take`. Both of these take a single `int` argument. As the name suggests, `Skip` discards the specified number of elements, and then returns everything else from its source. `Take` returns the specified number of elements from the start of the sequence and then discards the rest (so it is similar to `TOP` in SQL.)

There are predicate-driven equivalents, `SkipWhile` and `TakeWhile`. `SkipWhile` will discard items from the sequence until it finds one that matches the predicate, at which point it will return that and every item that follows for the rest of the sequence (whether or not the remaining items match the predicate). Conversely, `TakeWhile` returns items until it encounters the first item that does not match the predicate, at which point it discards that and the remainder of the sequence.

Although `Skip`, `Take`, `SkipWhile` and `TakeWhile` are all clearly order-sensitive, they are not restricted to just the ordered types such as `IOrderedEnumerable<T>`. They are also defined for `IEnumerable<T>`, which is reasonable because even though there may be no particular order guaranteed, any `IEnumerable<T>` always produces elements in some order. (The only way you can extract items from an `IEnumerable<T>` is one after another, so there will always be an order, even if it's meaningless.) Moreover, `IOrderedEnumerable<T>` is not widely implemented

outside of LINQ so it's quite common to have non-LINQ aware objects that produce items in a known order but which only implement `IEnumerable<T>`. These operators are useful in these scenarios, so the restriction is relaxed. Slightly more surprisingly, `IQueryable<T>` also supports these operations, but that's consistent with the fact that many database support `TOP` (roughly equivalent to `Take`) even on unordered queries. As always, individual providers may choose not to support individual operations, so in scenarios where there's no reasonable interpretation of these operators, they will just throw an exception.

A related operator is `DefaultIfEmpty<T>`. This returns the entire source collection, unless it's empty, in which case this returns a sequence containing a single item that has the default, zero-like value for `T`, i.e., null for a reference type, zero for numbers, etc.

Aggregation

The `Sum` and `Average` operators add together the values of all the source items. `Sum` returns the total, and `Average` returns the total divided by the number of items. They are available for collections of items of these numeric types: `decimal`, `double`, `float`, `int`, and `long`. There are also overloads that work with any item type, in conjunction with a lambda that takes an item and returns one of those numeric types. That allows us to write code such as Example 10-39 which works with a collection of `Course` objects, and calculates the average of a particular value extracted from the object: the course length in hours.

Example 10-39. Average operator with projection

```
Console.WriteLine("Average course length in hours: {0}",
    Course.Catalog.Average(course => course.Duration.TotalHours));
```

LINQ also defines `Min` and `Max` operators. You can apply these to any type of sequence, although it is not guaranteed to succeed—the particular provider you're using may report an error if it doesn't know how to compare the types you've used. For example, LINQ to Objects requires at least one of the objects in the sequence to implement `IComparable`.

`Min` and `Max` both have overloads that accept a lambda that gets the value to use from the source item. Example 10-40 uses this to find the date on which the most recent course was published.

Example 10-40. Max with projection

```
DateTime m = mathsCourses.Max(c => c.PublicationDate);
```

Notice that this does not return the course with the most recent publication date; it returns that course's publication date. If you want to select the object for which a particular property has the maximum value, you would use the `OrderByDescending` operator followed by `First`, or `FirstOrDefault`.

LINQ to Objects defines specialized overloads of `Min` and `Max` for sequences that return the numeric types that `Sum` and `Average` deal with, i.e. `decimal`, `double`, `float`, `int`, and `long`. It also defines similar specializations for the form that takes a lambda. These overloads exist to improve performance by avoiding boxing. The general-purpose form relies on `IComparable` and getting an interface type reference to a value always involves boxing that value. For large collections, boxing every single value would put considerable extra pressure on the garbage collector.

LINQ defines an operator called `Aggregate`, which generalizes the pattern that `Min`, `Max`, `Sum`, and `Average` all use, which is to produce a single result with a process that involves taking every source item into consideration. It's possible to implement all four of these operators in terms of `Aggregate`. Example 10-41 uses the `Sum` operator to calculate the total duration of all courses, and then uses the `Aggregate` operator to perform the exact same calculation.

Example 10-41. Sum and equivalent with Aggregate

```
double t1 = Course.Catalog.Sum(course => course.Duration.TotalHours);
double t2 = Course.Catalog.Aggregate(
    0.0, (hours, course) => hours + course.Duration.TotalHours);
```

Aggregation works by building up a value that represents what we know about all the items inspected so far, referred to as the *accumulator*. The type we use will depend on the knowledge that we want to accumulate. In this case, I'm just adding all the numbers together, so I'll use a `double` (because the `TimeSpan` type's `TotalHours` property is also a `double`).

Initially we have no knowledge, because we haven't looked at any items yet. We need to provide an accumulator value to represent this starting point, so the `Aggregate` operator's first argument is the *seed*, an initial value for the accumulator. In Example 10-41, the accumulator is just a running total, so the seed is `0.0`.

The second argument is a lambda that describes how to update the accumulator to incorporate information for a single item. Since my goal here is simply to calculate the total time, I just add the duration of the current course to the running total.

Once `Aggregate` has looked at every item, this particular overload returns the accumulator directly. It will be the total number of hours across all courses in this case.

The accumulator doesn't have to use addition. We can implement `Max`, using the same process, but a different accumulation strategy. Instead of maintaining a running total, the value representing everything we know so far about the data is simply the highest value seen yet. Example 10-42 shows the rough equivalent of Example 10-40. (It's not exactly the same, because Example 10-42 makes no attempt to detect an empty source. `Max` will throw an exception if this source is empty, but this will just return the date 0/0/0000.)

Example 10-42. Implementing Max with Aggregate

```
DateTime m = mathsCourses.Aggregate(
    new DateTime(),
   (date, c) => date > c.PublicationDate ? date : c.PublicationDate);
```

This illustrates that `Aggregate` does not impose any single meaning for the value that accumulates knowledge—the way you use it depends on what you're doing. Some operations require an accumulator with a bit more structure. Example 10-43 calculates the average course duration with `Aggregate`.

Example 10-43. Implementing Average with Aggregate

```
double average = Course.Catalog.Aggregate(
    new { TotalHours = 0.0, Count = 0 },
    (totals, course) => new
    {
        TotalHours = totals.TotalHours + course.Duration.TotalHours,
        Count = totals.Count + 1
    },
```

```
    totals => totals.TotalHours / totals.Count);
```

The average duration requires us to know two things: the total duration, and the number it items. So in this example, my accumulator uses a type that can contain two values, one to hold the total and one to hold the item count. I've used an anonymous type, but I could also have used `Tuple<double, int>`, or even written an ordinary type with a couple of properties. (In fact, a custom struct might have been a better choice, because it would have avoided allocating a new heap block for the accumulator at each iteration.)

> Example 10-43 relies on the fact that when two separate methods in the same component create instances of two structurally identical anonymous types, the compiler generates a single type that is used for both. The seed produces an instance of an anonymous type consisting of a `double` called `TotalHours` and an `int` called `Count`. The accumulation lambda also returns an instance of an anonymous type with the same member names and types in the same order. The C# compiler deems that these will in fact be the same type, which is important here, because `Aggregate` requires the lambda to accept and also return an instance of the accumulator type. If C# did not guarantee that the two expressions returning anonymous type instances in this example would return the exact same type, we could not depend on this code to compile correctly.

Example 10-43 uses a different overload than earlier example. It takes an extra lambda, which is used to extract the return value from the accumulator—the accumulator builds up the information I need to produce the result, but the accumulator itself is not the result in this example.

Of course, if all you want to do is calculate the sum, maximum, or average values, you wouldn't use `Aggregate`—you'd use the specialized operators designed to do those jobs. Not only are they simpler, they're often more efficient. (For example, a LINQ provider for a database might be able to generate a query that uses the database's built-in features to calculate the minimum or maximum value.) I just wanted to show the flexibility, using examples that are easily understood. But now that I've done that, Example 10-44 shows a particularly concise example of `Aggregate` that doesn't correspond to any other built-in operator. This takes a collection of rectangles, and returns the bounding box that contains all of those rectangles.

Example 10-44. Aggregating bounding boxes

```
public static Rect GetBounds(IEnumerable<Rect> rects)
{
    return rects.Aggregate(Rect.Union);
}
```

I'm using the `Rect` structure from the `System.Windows` namespace. This is part of WPF, but it's a very simple data structure that just contains four numbers—left, right, width, and height—so you can use it in non-WPF applications if you like.[1] Example 10-

[1] If you do so, be careful not to confuse it with another WPF type, `Rectangle`. That's an altogether more complex beast that supports animation, styling, layout, user input, data binding and

44 uses the `Rect` type's static `Union` method, which takes two `Rect` arguments, and returns a single `Rect` that is the bounding box of the two inputs (i.e. the smallest possible rectangle that contains both of the input rectangles).

I'm using the simplest overload of `Aggregate` here. It does the same thing as the one I used in Example 10-41, but it doesn't require me to supply a seed. It uses the type's default zero-like value as the seed. (So in fact I could have used this simpler overload in Example 10-41. The only reason I didn't was that I wanted to make the existence of a seed explicit in that example.) For classes, that would mean null, but `Rect` is a struct, so the seed is an instance of `Rect` with all its fields set to zero. Example 10-45 is equivalent to Example 10-44, I've just made the seed value explicit, and I've also written a lambda to invoke the method, instead of passing the method itself. With this sort of lambda that just passes its arguments straight on to an existing method, if you're using LINQ to Objects you can just pass the method name instead, so LINQ will call the target method directly rather than going through your lambda. (You can't do that with expression-based providers, because they require a lambda.) Using the method directly is more succinct and marginally more efficient, but it also makes for slightly harder to read code, which is why I've spelled it out in Example 10-45.

Example 10-45. More verbose and less obscure bounding box aggregation

```
return rects.Aggregate(new Rect(), (r1, r2) => Rect.Union(r1, r2));
```

These two examples work the same way. They start with a zero-sized rectangle as the seed. For the first item in the list, `Aggregate` will call `Rect.Union`, passing in the seed and the first rectangle. The `Union` method detects when one of its arguments is an empty rectangle, and just returns a copy of the first rectangle. That becomes the new accumulator value. Then for the next item in the source, `Aggregate` will pass `Union` the accumulator—a copy of the first rectangle—and the second rectangle, so the accumulator will become the bounding box of the first two rectangles. And that then gets passed to `Union` along with the third rectangle, and so on. Example 10-46 shows what the effect of this `Aggregate` operation would be if performed on a collection of four `Rect` values. (I've represented the four values here as `r1`, `r2`, `r3`, and `r4`. To pass them to `Aggregate`, they'd need to be inside a collection such as an array.)

Example 10-46. The effect of Aggregate

```
Rect bounds =
  Rect.Union(Rect.Union(Rect.Union(Rect.Union(new Rect(), r1), r2), r3), r4);
```

As I mentioned earlier, `Aggregate` is LINQ's name for an operation sometimes called reduce. You also sometimes see it called *fold*. LINQ went with the name `Aggregate` for the same reason it calls its projection operator `Select` instead of map, the more common name in functional programming languages: LINQ's terminology is more influenced by SQL than it is by academic languages.

---

various other WPF features. You would not want to attempt to use `Rectangle` outside of a WPF application.

Set Operations

LINQ defines three operators that use some common set operations to combine two sources. `Intersect` produces a result that contains only those items that were in both of the input sources. `Except` is the opposite: it includes only those items that were in one of the sources and not the other. The output of `Union` contains items that were in either (or both) of the input sources.

Although LINQ defines these set operations, most LINQ source types are not an exact abstraction of a set. With a mathematical set, any particular item either belongs to a set or it does not. There is no innate concept of the number of times a particular item appears in a set. `IEnumerable<T>` is not like that—it's a sequence of items, so it's possible to have duplicates, and the same is true of `IQueryable<T>`. This is not necessarily a problem, because some collections will happen never to get into a situation where they contain duplicates, and in some cases, the presence of duplicates won't cause a problem. However, it can sometimes be useful to take a collection that contains duplicates and remove them, leaving you with something that more closely resembles a set. For this, LINQ defines the `Distinct` operator, which removes duplicates. Example 10-47 contains a query that extracts the category names from all the courses, and then feeds that into the `Distinct` operator to ensure that each unique category name appears just once.

Example 10-47. Removing duplicates with Distinct

```
var categories = Course.Catalog.Select(c => c.Category).Distinct();
```

All of these set operators are available in two forms, because you can optionally pass any of them an `IEqualityComparer<T>`. This allows you to customize how the operators decide whether two items are the same thing.

Whole-Sequence Order-Preserving Operations

LINQ defines certain operators whose output includes every item from the source, and which preserve or reverse the order. Not all collections necessarily have an order, so these operators will not always be supported. However, LINQ to Objects supports all of them. The simplest is `Reverse`, which reverses the order of the elements.

The `Concat` operator combines two sequences. It returns a sequence which produces all of the elements from the first sequence, followed by all of the elements from the second sequence.

The `Zip` operator also combines two sequences, but instead of returning one after the other, it works with pairs of elements. So the first item it returns will be based on both the first item from the first sequence and the first item from the second sequence. The second item in the zipped sequence will be based on the second items from each of the sequences, and so on. The name `Zip` is meant to bring to mind how a zipper in an item of clothing brings two things together in perfect alignment. (It's not an exact analogy. When a zipper brings together the two parts, the teeth from the two halves interlock in an alternating fashion. But the `Zip` operator does not interleave its inputs like a physical zipper's teeth. It brings items from the two sources together in pairs.)

While `Reverse` and `Concat` just pass their items through unmodified, `Zip` works with pairs of items, and you need to tell it how you'd like them combined. So it takes a lambda with two arguments, and it will pass item pairs from the two sources as those arguments,

and will produce whatever your lambda returns as output items. Example 10-48 uses a selector that combines each pair of items using string concatenation.

Example 10-48. Combining lists with Zip

```
string[] firstNames = { "Ian", "Arthur", "Arthur" };
string[] lastNames = { "Griffiths", "Dent", "Pewty" };
IEnumerable<string> fullNames = firstNames.Zip(lastNames,
    (first, last) => first + " " + last);
foreach (string name in fullNames)
{
    Console.WriteLine(name);
}
```

The two lists that this example zips together contains first names and last names respectively. The output looks like this:

```
Ian Griffiths
Arthur Dent
Arthur Pewty
```

If the input sources contain different numbers of items, `Zip` will stop once it reaches the end of the smaller collection, and will not attempt to retrieve any further items from the longer collection.

The `SequenceEqual` operator bears a resemblance to `Zip`, in that it works on two sequences, and acts on pairs of items found at the same position in the two sequences. But instead of passing them to a lambda to be combined, `SequenceEqual` just compares each pair. If this comparison process finds that the two sources contain the same number of items, and that for every pair, the two items are equal, then it returns true. If the sources are of different lengths, or if even just one pair of items is not equal, it returns false. `SequenceEqual` has two overloads, one that accepts just the list with which to compare the source, and another that also takes an `IEqualityComparer<T>`, to customize what you mean by equal.

Grouping

Sometimes you will want to do more than just sorting items into a particular order. You may want to process all items that have something in common as a group. Example 10-49 uses a query to group courses by category, printing out a title for each category before listing all the courses in that category.

Example 10-49. Grouping query expressions

```
var subjectGroups = from course in Course.Catalog
                    group course by course.Category;

foreach (var group in subjectGroups)
{
    Console.WriteLine("Category: " + group.Key);
    Console.WriteLine();

    foreach (var course in group)
    {
        Console.WriteLine(course.Title);
    }
    Console.WriteLine();
```

```
}
```

A `group` clause takes an expression that determines group membership—in this case, any courses whose `Category` properties return the same value will be deemed to be in the same group. A `group` clause produces a collection in which each item implements `IGrouping<TKey, TItem>`, where `TKey` is the type of the grouping expression, and `TItem` is the input item type. (Since I'm using LINQ to Objects, and I'm grouping by category string, the type of the `subjectGroup` variable in Example 10-49 will be `IEnumerable<IGrouping<string, Course>>`.) This particular example produces three group objects, depicted in Figure 10-1.

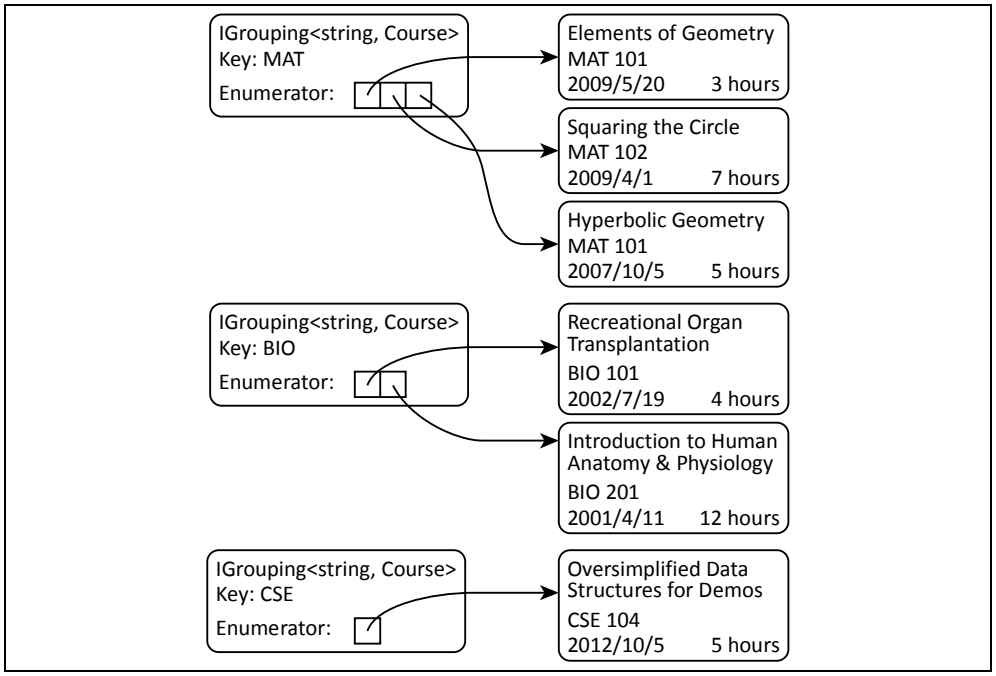

Figure 10-1. Result of evaluating a grouping query

Each of the `IGrouping<string, Course>` items has a `Key` property, and because the query grouped items by the course's `Category` property, each key contains a string value from that property. There are three different category names in the sample data in Example 10-17: `MAT`, `BIO`, and `CSE`, so these are the `Key` values for the three groups.

The `IGrouping<TKey, TItem>` interface derives from `IEnumerable<TItem>`, so each group object can be enumerated to find the items it contains. So in Example 10-49, the outer `foreach` loop iterates over the three groups returned by the query, and then the inner `foreach` loop iterates over the `Course` objects in each of the groups.

The query expression turns into the code in Example 10-50.

Example 10-50. Expanding a simple grouping query

```
var subjectGroups = Course.Catalog.GroupBy(course => course.Category);
```

Query expressions offer some variations on the theme of grouping. With a slight modification to the original query, we can arrange for the items in each group to be something other than the original `Course` objects. In Example 10-51, I've changed the

expression immediately after the `group` keyword from just `course` to `course.Title`.

Example 10-51. Group query with item projection

```
var subjectGroups = from course in Course.Catalog
                    group course.Title by course.Category;
```

This still has the same grouping expression, `course.Category`, so this produces three groups as before, but now it's of type `IGrouping<string, string>`. If you were to iterate over the contents of one of the groups, you'd find each group offers a sequence of strings, containing the course names. As Example 10-52 shows, the compiler expands this query into a different overload of the `GroupBy` operator.

Example 10-52. Expanding a group query with an item projection

```
var subjectGroups = Course.Catalog
    .GroupBy(course => course.Category, course => course.Title);
```

Query expressions are required to have either a `select` or a `group` as their final clause. However, if a query contains a `group` clause, that doesn't have to be the last clause. In Example 10-51, I modified how the query represents each item within a group (i.e., the boxes on the right of Figure 10-1) but I'm also free to customize the objects representing each group (the items on the left). By default I get the `IGrouping<TKey, TItem>` objects, but I can change this. Example 10-53 uses the optional `into` keyword in its `group` clause. This introduces a new range variable, which iterates over the group objects, which I can go on to use in the rest of the query. I could follow this with other clause types such as `orderby` or `where`, but in this case I've chosen to use a `select` clause.

Example 10-53. Group query with group projection

```
var subjectGroups = from course in Course.Catalog
                    group course by course.Category into category
                    select string.Format("Category '{0}' contains {1} courses",
                        category.Key, category.Count());
```

The result of this query is an `IEnumerable<string>`, and if you print out all the strings it produces, you get this:

```
Category 'MAT' contains 3 courses
Category 'BIO' contains 2 courses
Category 'CSE' contains 1 courses
```

As Example 10-54 shows, this expands into a call to the same `GroupBy` overload that Example 10-50 uses, and then uses the ordinary `Select` operator for the final clause.

Example 10-54. Expanded group query with group projection

```
IEnumerable<string> subjectGroups = Course.Catalog
    .GroupBy(course => course.Category)
    .Select(category => string.Format("Category '{0}' contains {1} courses",
                                      category.Key, category.Count()));
```

LINQ defines some more overloads for the `GroupBy` operator that are not accessible from the query syntax. Example 10-55 shows an overload that provides a slightly more direct equivalent to Example 10-53.

Example 10-55. GroupBy with key and group projections

```
IEnumerable<string> subjectGroups = Course.Catalog.GroupBy(
    course => course.Category,
    (category, courses) => string.Format("Category '{0}' contains {1} courses",
                                         category, courses.Count()));
```

This overload takes two lambdas. The first is the expression by which items are grouped. The second is used to produce each group object. Unlike the previous examples, this does not use the `IGrouping<TKey, TItem>` interface. Instead, the final lambda receives the key as one argument, and then a collection of the items in the group as the second. This is exactly the same information that `IGrouping<TKey, TItem>` encapsulates, but because this form of the operator can pass these as separate arguments, it removes the need for objects to represent the groups.

There's yet another version of this operator shown in Example 10-56. It combines the functionality of all the other flavors.

Example 10-56. GroupBy operator with key, item, and group projections

```
IEnumerable<string> subjectGroups = Course.Catalog.GroupBy(
    course => course.Category,
    course => course.Title,
    (category, titles) =>
         string.Format("Category '{0}' contains {1} courses: {2}",
                       category, titles.Count(), string.Join(", ", titles)));
```

This overload takes three lambdas. The first is the expression by which items are grouped. The second determines how individual items in a group are represented—this time I've chosen to extract the course title. The third lambda is used to produce each group object, and as with Example 10-55, this final lambda is passed the key as one argument, and its other argument gets the group items, as transformed by the second lambda. So rather than the original `Course` items, this second argument will be an `IEnumerable<string>` containing the course titles, because that's what the second lambda in this example requested. The result of this `GroupBy` operator is once again a collection of strings, but now it looks like this:

```
Category 'MAT' contains 3 courses: Elements of Geometry, Squaring the Circle, Hy
perbolic Geometry
Category 'BIO' contains 2 courses: Recreational Organ Transplantation, Introduct
ion to Human Anatomy and Physiology
Category 'CSE' contains 1 courses: Oversimplified Data Structures for Demos
```

I've shown four versions of the `GroupBy` operator. All four take a lambda that selects the key to use for grouping, and the simplest overload takes nothing else. The others let you control the representation of individual items in the group, or the representation of each group, or both. In fact there are four more versions of this operator. They offer all the same services as the four I've shown already, but also take an `IEqualityComparer<T>`, which lets you customize the logic that decides whether two keys are considered to be the same for grouping purposes.

There is one other operator that groups its outputs, called `GroupJoin`, but it does so as part of a join operation.

Joins

LINQ defines a `Join` operator that provides a way to use related data from some other source, much as a database query can join information from one table with data in another table. Suppose our application stored a list of which students had signed up for which courses. If you stored that information in a file, you wouldn't want to copy the full details for either the course or the student out into every line—you'd want just enough information to identify a student and a particular course. In my example data, courses are uniquely identified by the combination of the category and the number. So to record who's signed up for what, we'd need records containing three pieces of information: the course category, the course number, and something to identify the student. The class in Example 10-57 shows how we might represent such a record in memory.

Example 10-57. Class associating a student with a course

```
public class CourseChoice
{
    public int StudentId { get; set; }

    public string Category { get; set; }

    public int Number { get; set; }
}
```

In your application, once you've loaded this information into memory, you may want access to the `Course` objects, rather than just the information identifying the course. We can get this with a `join` clause, as shown in Example 10-58 (which also supplies some additional sample data using the `CourseChoice` class, so that the query has something to work with).

Example 10-58. Query with join clause

```
CourseChoice[] choices =
{
    new CourseChoice { StudentId = 1, Category = "MAT", Number = 101 },
    new CourseChoice { StudentId = 1, Category = "MAT", Number = 102 },
    new CourseChoice { StudentId = 1, Category = "MAT", Number = 207 },
    new CourseChoice { StudentId = 2, Category = "MAT", Number = 101 },
    new CourseChoice { StudentId = 2, Category = "BIO", Number = 201 },
};

var studentsAndCourses = from choice in choices
                         join course in Course.Catalog
                           on new { choice.Category, choice.Number }
                           equals new { course.Category, course.Number }
                         select new { choice.StudentId, Course = course };

foreach (var item in studentsAndCourses)
{
    Console.WriteLine("Student {0} will attend {1}",
        item.StudentId, item.Course.Title);
}
```

This prints out one line for each entry in the choices array. It shows the title for each course, because even though that was not available in the input collection, the `join`

clause located the relevant item in the course catalog. Example 10-59 shows how the compiler translates the query in Example 10-58.

Example 10-59. Using the Join operator directly

```
var studentsAndCourses = choices.Join(
    Course.Catalog,
    choice => new { choice.Category, choice.Number },
    course => new { course.Category, course.Number },
    (choice, course) => new { choice.StudentId, Course = course });
```

The `Join` operator's job is to find an item in the second sequence that corresponds to the item in the first. This correspondence is determined by the first two lambdas—items from the two sources will be considered to correspond to one another if the values returned by these two lambdas are equal. This example uses an anonymous type, and depends on the fact that two structurally identical anonymously typed instances in the same assembly share the same type. In other words, those two lambdas both produce objects with the same type. The compiler generates an `Equals` method for any anonymous type that compares each member in turn, so the effect of this code is that two rows are considered to correspond if their `Category` and `Number` properties are both equal.

I've set this example up so that there can only be one match, but what would happen if the course category and number did not uniquely identify a course for some reason? If there are multiple matches for any single input row, the `Join` operator will produce one output item for each match, so in that case, we'd get more output items than there were entries in the `choices` array. Conversely, if an item in the first source has no corresponding item in the second collection, `Join` will not produce any output for the item—it effectively ignores that input item.

LINQ offers an alternative join type that handles input rows with either zero, or multiple corresponding rows differently than the `Join` operator. Example 10-60 shows the modified query expression. (The difference is the addition of `into courses` on the end of the `join` clause, and the final `select` clause refers to that instead of the `course` range variable.) This produces output in a different form, so I've also modified the code that prints out the results.

Example 10-60. A grouped join

```
var studentsAndCourses =
    from choice in choices
    join course in Course.Catalog
      on new { choice.Category, choice.Number }
      equals new { course.Category, course.Number } into courses
    select new { choice.StudentId, Courses = courses };

foreach (var item in studentsAndCourses)
{
    Console.WriteLine("Student {0} will attend {1}",
        item.StudentId,
        string.Join(",", item.Courses.Select(course => course.Title)));
}
```

As Example 10-61 shows, this causes the compiler to generate a call to the `GroupJoin` operator, instead of `Join`.

Example 10-61. GroupJoin operator

```
var studentsAndCourses = choices.GroupJoin(
    Course.Catalog,
    choice => new { choice.Category, choice.Number },
    course => new { course.Category, course.Number },
    (choice, courses) => new { choice.StudentId, Courses = courses });
```

This form of join produces one result for each item in the input collection by invoking the final lambda. Its first argument is the input item, and its second argument will be a collection of all the corresponding objects from the second collection. (Compare this with `Join`, which invokes its final lambda once for each match, passing the corresponding items one at a time.) This provides a way to represent an input item that has no corresponding items in the second collection: the operator can just pass an empty collection.

Both `Join` and `GroupJoin` also have overloads that accept an `IEqualityComparer<T>`, so that you can define a custom meaning for equality for the values returned by the first two lambdas.

Conversion

Sometimes you will need to convert a query of one type to some other type. For example, you might have ended up with a collection where the type argument specifies some base type (e.g., `object`), but where you have good reason to believe that the collection actually contains items of some more specific type (e.g., `Course`). When dealing with individual objects, you can just use the C# cast syntax to convert the reference to the type you believe you're dealing with. Unfortunately, this doesn't work for types such as `IEnumerable<T>` or `IQueryable<T>`.

Although covariance means that an `IEnumerable<Course>` is implicitly convertible to an `IEnumerable<object>`, you cannot convert in the other direction even with an explicit downcast. If you have a reference of type `IEnumerable<object>`, attempting to cast that to `IEnumerable<Course>` will only succeed if the object implements `IEnumerable<Course>`. It's quite possible to end up with a sequence that consists entirely of `Course` objects but which does not implement `IEnumerable<Course>`. Example 10-62 creates just such a sequence, and it will throw an exception when it tries to cast to `IEnumerable<Course>`.

Example 10-62. How not to cast a sequence

```
IEnumerable<object> sequence = Course.Catalog.Select(c => (object) c);
var courseSequence = (IEnumerable<Course>) sequence; // InvalidCastException
```

This is a contrived example of course. I forced the creation of an `IEnumerable<object>` by casting the `Select` lambda's return type to `object`. However, it's easy enough to end up in this situation for real, in only slightly more complex circumstances. Fortunately, there's an easy solution. You can use the `Cast<T>` operator, shown in Example 10-63.

Example 10-63. How to cast a sequence

```
var courseSequence = sequence.Cast<IEnumerable<Course>>();
```

This returns a query that produces every item in its source in order, but it casts each item to the specified target type as it does so. This means that although the initial `Cast<T>` might succeed, it's possible that you'll get an `InvalidCastException` some point

later when you try to extract values from the sequence. After all, in general, the only way the `Cast<T>` operator can verify that the sequence you've given it really does only ever produce values of type `T` is to extract all those values and attempt to cast them. It can't evaluate the whole sequence up front because you might have supplied an infinite sequence. How is it to know whether the first billion items your sequence produces will be of the right type, but after that you return one of an incompatible type? So its only option is to try casting items one at a time.

> `Cast<T>` and `OfType<T>` look similar, and developers sometimes use one when they should have used the other (usually because they didn't know both existed). `OfType<T>` does almost the same thing as `Cast<T>`, but it silently filters out any items of the wrong type instead of throwing an exception. If you expect and want to ignore items of the wrong type, use `OfType<T>`. If you do not expect items of the wrong type to be present at all, use `Cast<T>`, because if you turn out to be wrong, it will let you know by throwing an exception, reducing the risk of allowing a potential bug to remain hidden.

LINQ to Objects defines an `AsEnumerable<T>` operator. This just returns the source without modification—it does nothing. Its purpose is to force the use of LINQ to Objects even if you are dealing with something that might have been handled by a different LINQ provider. For example, suppose you have something that implements `IQueryable<T>`. That interface derives from `IEnumerable<T>`, but the extension methods that work with `IQueryable<T>` will take precedence over the LINQ to Objects ones. If your intention is to execute a particular query on a database, and then use further client-side processing of the results with LINQ to Objects, you can use `AsEnumerable<T>` to draw a line that says: this is where we move things to the client side.

Conversely, there's also `AsQueryable<T>`. This is designed to be used in scenarios where you have a variable of static type `IEnumerable<T>` that you believe might contain a reference to an object that also implements `IQueryable<T>`, and you want to ensure that any queries you create use that instead of LINQ to Objects. If you use this operator on a source that does not in fact implement `IQueryable<T>`, it returns a wrapper that implements `IQueryable<T>` but which uses LINQ to Objects under the covers.

Yet another operator for selecting a different flavor of LINQ is `AsParallel`. This returns a `ParallelQuery<T>`, which lets you build queries to be executed by Parallel LINQ (PLINQ). I will discuss PLINQ in Chapter 17.

There are some operators that convert the query to other types, and which also have the effect of executing the query immediately, rather than building a new query chained off the back of the previous one. `ToArray` and `ToList` return an array or a list respectively, containing the complete results of executing the input query. `ToDictionary` and `ToLookup` do the same but rather than producing a straightforward list of the items, they both produce results that support associative lookup. `ToDictionary` returns an `IDictionary<TKey, TValue>`, so it is intended for scenarios where a key corresponds to exactly one value. `ToLookup` is designed for scenarios where a key may be associated with multiple values, so it returns a different type `ILookup<TKey, TValue>`. I did not mention this interface in Chapter

5 because it is specific to LINQ. It is essentially the same as the dictionary interface, except the indexer returns an `IEnumerable<TValue>` instead of a single `TValue`.

While the array and list conversions take no arguments, the dictionary and lookup conversions need to be told what value to use as the key for each source item. You tell it by passing a lambda, as Example 10-64 shows. This uses the course's `Category` property as the key.

Example 10-64. Creating a lookup

```
ILookup<string, Course> categoryLookup =
    Course.Catalog.ToLookup(course => course.Category);
foreach (Course c in categoryLookup["MAT"])
{
    Console.WriteLine(c.Title);
}
```

The `ToDictionary` operator offers an overload that takes the same argument and returns a dictionary. It would throw an exception if you called in the same way that I called `ToLookup` in Example 10-64, because multiple course objects share categories, so they would map to the same key. `ToDictionary` requires each object to have a unique key. To produce a dictionary from the course catalog, you'd either need to group the data by category first, and have each dictionary entry refer to an entire group, or you'd need a lambda that returned a composite key based on both the course category and number, because that combination is unique to a course.

Both operators also offer an overload that takes a pair of lambdas, one that extracts the key, and a second that chooses what to use as the corresponding value—you are not obliged to use the source item as the value. Finally, there are overloads that also take an `IEqualityComparer<T>`.

Sequence Generation

The `Enumerable` class defines the extension methods for `IEnumerable<T>` that comprise LINQ to Objects. It also offers a few additional (non-extension) static methods that can be used to create new sequences. `Enumerable.Range` takes two `int` arguments, and returns an `IEnumerable<int>` that produces a sequentially increasing series of numbers starting from the value of the first argument, that's as long as the second argument. For example, `Enumerable.Range(15, 10)` produces a sequence containing the numbers 15 to 24 (inclusive).

`Enumerable.Repeat<T>` takes a value of type `T` and a count. It returns a sequence that will produce that value the specified number of times.

`Enumerable.Empty<T>` returns an `IEnumerable<T>` that contains no elements. This may not sound very useful, because there's a much less verbose alternative. You could write `new T[0]`, which creates an array that contains no elements. (Arrays of type `T` implement `IEnumerable<T>`.) In fact, that's exactly what the current implementation of `Enumerable.Empty<T>` appears to return, although you should not depend on it being an array because that's not documented. However, the advantage of `Enumerable.Empty<T>` is that for any given `T`, it returns the same instance every time. This means that if for any reason you end up needing an empty sequence repeatedly

in a loop that executes many iterations, `Enumerable.Empty<T>` is more efficient because it puts less pressure on the garbage collector.

Other LINQ implementations

Most of the examples I've shown in this chapter have used LINQ to Objects, except for a handful that have referred to LINQ to Entities, a provider used with databases. In this final section I will provide a quick description of some other LINQ-based technologies. This is not a comprehensive list, because anyone can write a LINQ provider.

Entity Framework

The database examples I have shown have used LINQ to Entities, which is part of the Entity Framework (EF). The EF is a data access technology that ships as part of the .NET Framework that can map between a database and an object layer. It supports multiple database vendors. I will describe the EF in more detail in Chapter 19. For this chapter, it is interesting because it is one of the most widely used LINQ providers.

The EF relies on `IQueryable<T>`. For each persistent entity type in a data model, the EF can provide an object that implements `IQueryable<T>`, and which can be used as the starting point for building queries to retrieve entities of that type and of related types. Since `IQueryable<T>` is not unique to the EF, you will be using the standard set of extension methods provided by the `Queryable` class in the `System.Linq` namespace, but that mechanism is designed to allow each provider to plug in its own behavior.

Because `IQueryable<T>` defines the LINQ operators as methods that accept `Expression<T>` arguments and not plain delegates types, any expressions you write in either query expressions or as lambda arguments to the underlying operator methods will turn into compiler-generated code that creates a tree of objects representing the structure of the expression. The EF relies on this to be able to generate database queries that fetch the data you require. This means that you are obliged to use lambdas—unlike with LINQ to Objects, you cannot use anonymous methods or delegates with an EF query.

Because `IQueryable<T>` derives from `IEnumerable<T>`, it's possible to use LINQ to Objects operators on any EF source. You can do this explicitly with the `AsEnumerable<T>` operator, but it could also happen accidentally if you used an overload that's supported by LINQ to Objects and not `IQueryable<T>`. For example, if you attempt to use a delegate instead of a lambda as, say, the predicate for the `Where` operator, this will fall back to LINQ to Objects, and the upshot of that is that LINQ to Entities will end up downloading the entire contents of the table and then evaluating the `Where` operator on the client side. This is unlikely to be a good idea.

LINQ to SQL

LINQ to SQL is another data access technology. Unlike the EF, it is designed specifically for Microsoft's SQL Server. It has a slightly different philosophy: it is designed as a convenient .NET API for accessing information in a database rather than as a layer between your database and your objects, so it does not have extensive features for

mapping between the structure of data in your database, and the design of your domain model.

LINQ to SQL presents objects representing specific tables in the database. These table objects implement `IQueryable<T>`, so when it comes to writing queries, LINQ to SQL works in a similar way to the EF.

WCF Data Services Client

WCF Data Services provide the ability to present and consume data over HTTP, using the standard Open Data Protocol (OData). This presents data using either XML or JSON, and defines a way to express queries that include filtering, ordering, and joining. The client-side part of this technology includes an `IQueryable<T>`-based LINQ provider. However, it supports only a fairly small subset of the standard LINQ operators, because the OData standard only makes it possible to encode a fairly limited range of queries.

Parallel LINQ (PLINQ)

Parallel LINQ is similar to LINQ to Objects, in that it is based on objects and delegates rather than expression trees and query translation. But when you start asking for results from a query, where possible it will use multithreaded evaluation, using the thread pool to try and use the available CPU resources efficiently. Chapter 17 will show PLINQ in action.

LINQ to XML

LINQ to XML is not a LINQ provider. I'm mentioning here because its name makes it sounds like one. It's really an API for creating and parsing XML documents. It's called LINQ to XML because it was designed to make it easy to execute LINQ queries against XML documents, but it achieves this by presenting XML documents through a .NET object model, and providing methods that extract features from the document in terms of `IEnumerable<T>`. This enables it to defer to LINQ to Objects to define and execute the queries.

Reactive Extensions

The .NET Reactive Extensions (or Rx, as they're often abbreviated) are the subject of the next chapter so I won't say too much about them here, but they are a good illustration of how LINQ operators can work on a variety of different types. Rx inverts the model shown in this chapter where we ask a query for items once we're good and ready. So instead of writing a `foreach` loop that iterates over query, or calling one of the operators that evaluates the query such as `ToArray` or `SingleOrDefault`, an Rx source calls us when it's ready to supply data.

Despite this inversion, there is a LINQ provider for Rx which supports most of the standard LINQ operators.

Summary

In this chapter, I showed the query syntax that supports some of the most commonly used LINQ features. This lets us write queries in C# that resemble database queries, but which can query any LINQ provider, including LINQ to Objects, which lets us run queries against our object models. I showed the standard LINQ operators for querying, all of which are available with LINQ to Objects, and most of which are available with database providers. I also provided a quick roundup of some of the common LINQ providers for .NET applications.

The last provider I mentioned was Rx. But before we look at Rx's LINQ provider, the next chapter will begin by looking at how Rx itself works.